PureConnect®

2019 R4

Generated:

06-February-2020

Content last updated:

05-February-2020

See Change Log for summary of changes.

GENESYS

Interaction Scripter

Developer's Guide

Abstract

This publication is for Web Page Developers who create sophisticated campaign scripts using Advanced Interaction Scripter. Experience with ECMA-262 JavaScript is recommended as a prerequisite.

For the latest version of this document, see the PureConnect Documentation Library at: http://help.genesys.com/cic.

Table of Contents

What is Interaction Scripter?

Interaction Scripter is a client application that executes campaign scripts written in HTML and Javascript. This application is used primarily by call center agents. Scripter's primary job is to execute *campaign scripts*—HTML pages that guide call center agents through stages of a campaign call. Scripts are displayed inside an embedded instance of Microsoft's Internet Explorer web browser. Since the browser component is built-in, Interaction Scripter can intercept and respond to web events. It fully leverages IE's feature set and can render sophisticated user interfaces.

Interaction Scripter allows call center agents to login to running campaigns. Each campaign has an associated campaign script (written in HTML) that implements screen pop, navigational aids, narratives, and data entry forms. Scripts are rendered by an embedded web browser component. Information that was collected or modified by agents is routed back to campaign databases.

> **Note**: The term "Interaction Scripter" refers to the client application that executes scripts, and also to the programming API that is used to create scripts. This documentation uses "Interaction Scripter Client" or "Scripter Client" to differentiate between the client application and the Interaction Scripter programming language. For details of running Interaction Scripter Client from a user perspective, refer to the *Interaction Scripter Client User Guide*.

Navigating a campaign script is much like surfing the Internet. Agents use simple point-and-click mouse movements, rather than typed commands, to control the Interaction Scripter Client. Each screen tells the agent what to say, and offers appropriate options for that stage of the call. Campaign scripts can display any combination of text, graphics, and visual controls. Interaction Scripter is a *scripting host* that exposes specialized telephony objects to scripting languages.

Interaction Scripter client is a powerful, stable, extensible, customizable DHTML client that uses off-the-shelf Internet technology to deliver user interfaces that proprietary scripting clients cannot match. Interaction Scripter's unique combination of API and client features provides a world-class solution that offers immense power.
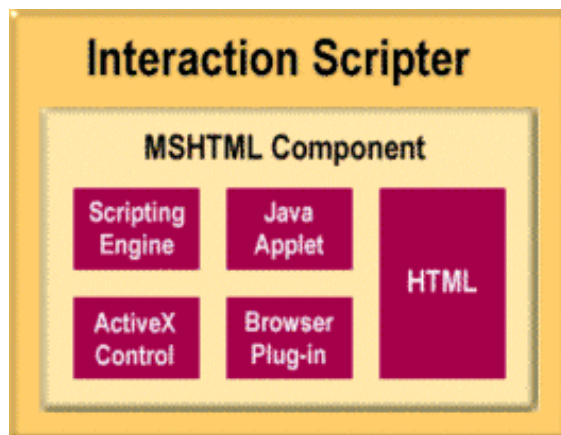
Interaction Scripter can be fully customized using nothing more than standard HTML code. However, its power and extensibility become apparent when one considers that Scripter fully supports Dynamic HTML (DHTML) and ECMA-262 JavaScript. These scripting languages, although not required for the deployment of the client, allow developers to greatly extend the functionality of campaign scripts by manipulating call, chat, conference, user, or queue objects in campaign scripts.

- Campaign scripts are created using well-understood and well-documented HTML coding technologies. Scripts can be created using off-the-shelf HTML development tools.   The pervasiveness of individuals skilled in writing and maintaining HTML code makes script development easy and inexpensive.
- Campaign scripts run in the Interaction Scripter .NET Client. You cannot run scripts in a web browser, since the client is a scripting host that exposes methods and properties that browsers do not provide. The Scripter Client offers state-of-the-art browsing, since an instance of Microsoft's Internet Explorer is embedded into the client application.

Interaction Scripter Client supports two types of scripts

All campaigns except agentless campaigns have a script that displays contact information to an agent. Scripts prompt for data entry, tell agents what to say, and provide navigation options that agents use to transition between calls or stages of a call. Interaction Scripter .NET Client populates the agent's display with information pertaining to the call, the customer, and the campaign, based on behavior defined in a script. Information collected or modified by agents is routed back to campaign database tables.

Interaction Scripter Client supports two types of scripts: **base scripts** configured by non-programmers in Interaction Dialer Manager, and **custom scripts** created by web developers.

- *Base scripts* (also called *Standard Forms*) display columns from the contact list, and may offer textual prompts for the Dialer agent to read to the contact. Base scripts provide simple page-to-page navigation controls and call disposition options.
- *Custom scripts* also provide screen pop, navigation controls and disposition options, but with any desired appearance and layout. Custom scripts require web development expertise to create.

Interaction Scripter Client supports any combination of base and custom scripts when multiple campaigns are running. Agents receive exactly the information and options they need to process each call.

Scripter from a programmer's perspective

From a programming perspective, scripts written for Interaction Scripter client support events, actions, attributes, and behaviors. These elements manipulate objects in all environments, regardless of whether Interaction Dialer is also installed. *Predictive* actions, events, and attributes work only when Interaction Dialer is installed on the CIC server. *Standard* actions, events, and attributes work with or without Interaction Dialer.

Event

Events are notification messages from CIC server that trigger script functions. For example, an event can provide notification that a queue on the server has changed. When a call is placed on a queue, this event changes the queue, generating an event message.

Predictive events are notification events associated with campaign activities for Predictive, Power or Preview campaigns. Predictive events are raised by Scripter when an agent is logged into Dialer. All predictive events are functions declared in a script that are called when an event occurs.

Standard events are generalized and can be used in any script, including scripts for blended environments. These events are not directly associated with being logged into Dialer, though they can be used when logged into Dialer too. Standard events are generated when queues change.

Actions

Actions are messages from Interaction Scripter scripts to the server that trigger an action on the CIC server.

Standard Actions perform normal operations on phone calls, such as picking up a call, placing it on hold, transferring, or recording. Standard actions provide basic telephony integration with the PureConnect platform. These actions allow scripts to manipulate telephone calls.

Predictive Actions attributes and events are only valid if Interaction Dialer is installed on the CIC Server and the user is logged into Dialer. Predictive actions are only applicable in Interaction Dialer campaigns, and can be applied only to campaign calls. Predictive actions are also useful in preview mode, when information about a party is pushed to an agent before the agent initiates the call.

Attributes

Attributes are data items passed by actions to the CIC server. A Dialer attribute is data from a column in a database that is associated with a campaign. Every Dialer database column is automatically associated with a script object of the same name with IS_Attr_ prefixed. For example, the database column "address" is available as script attribute "IS_ATTR_ADDRESS".

If the attribute is first declared in the script, it will go back to the Dialer server during a call complete function, and it can be accessed from a handler. In the CIC environment, an attribute is a piece of information about an object (such as a telephone call) that travels with the object. An example might be the telephone number of the individual called during a campaign. The server passes attributes to the client application when a new call event occurs. The client passes attributes to the server when a call-complete action is performed.

Predictive attributes are attributes that are normally used with either a Predictive, Preview or Power dialing campaigns. These attributes are not to be used in blended environments, for example in inbound pages loaded in scripter. The predictive base view for dialer is not loaded in an inbound page, thus these attributes would not return any values.

System Services attributes are supplemental predictive attributes that retrieve information about a Dialer agent, such as the agent's name, ID, or client status. System services are read-only.

Custom attributes are also supported. Scripter provides the ability to create any attribute within a custom script. These attributes can be references to the actual values in the call list or can be a newly created attribute declared in a meta tag within the pages loaded in scripter.

Behaviors

Behaviors are like command line parameters and are used to change the way that Scripter behaves when running custom scripts. This means that the behaviors are limited to the custom scripts that implement them rather than applying globally.

Predictive behaviors are only applicable for custom scripts associated with a Predictive, Power or Preview campaign.

Writing custom scripts for Interaction Connect or Scripter .NET

Starting with PureConnect 2018 R3, Dialer agents can use **Interaction Connect** or **Interaction Scripter .NET** to process outbound calls that use custom scripts. As before, both clients support base scripts.

Base scripts run in both clients

- Base scripts do not require programming expertise to create. With appropriate access rights, anyone can use the Scripts container in Interaction Administrator to design a base script.
- Every base script is compatible with Interaction Connect and Interaction Scripter .NET. The same base script may be used by a pool of agents running either or both applications.

Custom scripts run in one client or the other

Programmers develop custom scripts using HTML and JavaScript, in accordance with the *Interaction Scripter Developer Guide*. The script appears on an agent's screen when a call is previewed or routed to the agent.

Scripter .NET and Connect process JavaScript statements differently–either synchronously or asynchronously. This requires developers to implement different coding techniques. The resulting custom script is compatible with one client or the other. A custom script cannot be compatible with both clients.

Generally, web applications that perform tasks concurrently (asynchronously) are faster and more responsive than applications that perform tasks consecutively (synchronously). A synchronous application waits for something to finish before moving to another task. An asynchronous application starts the next task before a previous task finishes.

Scripter .NET is synchronous

Scripter .NET executes one JavaScript statement at a time, waiting for each consecutive statement to complete before moving to the next.

Suppose that you want Scripter .NET to transfer a call and then print a message. The pseudocode below uses the IS_Action_Transfer action. Since Scripter .NET is synchronous, it executes the IS_Action_Transfer action, waits until that action is complete, and then prints a message.

```
IS_Action_Transfer(CallId, false, "222-3333");
print('Transfer Success');
```

Interaction Connect is asynchronous

Interaction Connect executes JavaScript statements concurrently and independently from one another.

To tailor a script for Connect, programmers must ensure that the next operation does not begin until a signal is received indicating that the previous operation has completed. This is accomplished by implementing a custom callback (if required by a method) or by using a callback property of an IS_Action, which will be explained in a moment. When the function completes, the callback receives a signal telling it when to execute subsequent statements in its code block.

In Connect, the example above would not execute as the developer intended. Connect would execute both statements concurrently. Without a callback to tell Connect that the Transfer action is completed, it has no way to know that the developer's intent is to print a message after the transfer is completed.

To wait until the transfer ends, pass a callback function as a method parameter. In the code block for the callback, add any statements you want to execute *after* the calling method completes.

The callback property ensures that IS_Actions execute properly in Interaction Connect

Starting with 2018 R3, all [Interaction Scripter actions](#) (IS_Actions) provide a callback property for use in Connect scripts only. Each custom script action has a new callback attribute named "callback". If the action fails, the callback will be invoked with an error. If the action is successful, the callback will be invoked with no error. Customers can use this to determine how to proceed in the event that the action failed or completed successfully. For example:

```
IS_Action_Transfer.callback = function(error) {
    if(error) {
        console.error("IS_Action_Transfer failed");
    } else {
        console.log("IS_Action_Transfer performed successfully");
    }
}
```

In practice, you'll want to define a function to pass to the action any attributes it needs. In the example below, IS_Action_Transfer's callback property won't execute statements in its function code block until the transfer completes.

```
function IS_Action_Transfer() {
  IS_Action_Transfer.callid = 1234;
  IS_Action_Transfer.consult = False;
  IS_Action_Transfer.recipient = '377-522-2222';
  IS_Action_Transfer.callback = function(error) {
    if (error) {
      console.error("The Transfer action failed.");
    } else {
      console.log("The Transfer action was a success");
    }
  }
}
```

This is the key to writing scripts that run asynchronously in Interaction Connect. Statements inside the callback function block (highlighted above) execute only after the action completes. The callback will return an error if the action fails. If the action was successful, no error is returned.

> In this documentation, callbacks that apply exclusively to Interaction Connect are individually noted.

Additional callbacks added to support Interaction Connect

In addition to callbacks for actions, you can use callbacks designed for scripting objects—such as calls and chats.

CallObject.callObjectInitializedHandler	Allows a script to dial a number after waiting asynchronously for a call object to be created.
ChatObject.requestedAttributeReturnHandler	Allows a script to asynchronously get a chat attribute by first setting this callback and then calling the chatObject.getAttribute method.
ChatObject.chatObjectInitializedHandler	Allows a script to start a chat after waiting asynchronously for a ChatObject to be created.
ConferenceObject.conferenceObjectInitializedHandler	This callback is invoked when the conference object has initialized.
ConferenceObject.conferenceStartedHandler	This callback is invoked when the conference call has started.

Other scripting modifications to support Interaction Connect

- The IS_Action_Transfer action supports an "audience" parameter, used to toggle between different parties on a consult transfer call.
- A new standard action, IS_Action_CompleteConsult ends a consult call in scripts for Interaction Connect only.
- IS_Action_CallComplete has a new Boolean parameter named makefollowupcall. It indicates whether the user should be put into "Additional Follow Up status", in support of a feature that allows an agent to dial additional calls while in that status.
- For custom scripts in Scripter Connect, the wav file specified for the IS_Action_PlayWav action must be located in the Resource Path directory on the CIC server (I3\IC\Resources by default).
- The Queue.connect method is now asynchronous, meaning that it can accept a single optional callback argument that takes no parameters. In addition, Queue.connect no longer supports Line Queue as a Type parameter. Scripts for Scripter .NET do not need to specify a callback, but scripts for Connect must specify it.
- The Queue.startCallObjectsEnum, Queue.startChatObjectsEnum, Queue.startConferenceObjectsEnum and Queue.startObjectIdsEnum properties now accept an optional callback (for use with Connect scripts only) whose single parameter contains the result. User.startAccessibleQueuesEnum and User.startViewableWorkgroupsEnum work the same way.
- See Revisions for a complete list of changes to this documentation.

Summary

Since Interaction Connect is a web application, it runs in a browser without the need to install software on agent desktops. The option to replace a desktop application with a cloud application is compelling. As a prerequisite to using Connect, customers must manually update legacy scripts (or write new scripts) to run asynchronously.

- Legacy scripts written for Interaction Scripter .NET work as before in that client application.
- Due to the general complexity of script programming, Genesys cannot provide a script migration utility. Fortunately, most JavaScript developers are familiar with callbacks. Implementing new callbacks to support asynchronous execution should not present an obstacle for most customers.
- A custom script must be written for one client or the other.
- The same custom script cannot be used by both Scripter .NET and Interaction Connect.
- Scripts for Interaction Connect implement callback functions that wait for asynchronous operations complete.
- Scripts for Scripter .NET should not implement callbacks designed for Connect.

See Also

Sample Interaction Connect scripts

Sample Interaction Connect scripts

Although Interaction Connect and Scripter .NET provide similar functionality, Interaction Connect uses JavaScript promises to execute tasks asynchronously, while Scripter .NET follows a synchronous programming model. This article contains downloadable sample scripts that show how to carry out custom scripting tasks for Interaction Connect in a way that works with Scripter .NET's synchronous behavior.

It's important to keep this difference between synchronous and asynchronous programming in mind when you're creating scripts for Interaction Connect, so you can avoid potentially undesirable side effects. Consider the code below, which sends a disposition request at the end of a call and attempts to set the agent back to "Available" status.

```java
{code:java}
function callComplete(disposition) {
    // Assign the disposition to the action.
    IS_Action_CallComplete.wrapupcode = disposition;


    // Send the call disposition.
    IS_Action_CallComplete.click();


    // Set the agent back to Available.
    IS_Action_ClientStatus = "Available";
    IS_Action_ClientStatus.click();
}
{code}
```

If this code is run in Scripter .NET, each action is executed consecutively. The disposition request is sent and execution halts until the disposition request has completed. Once the disposition request has completed, code execution continues with the next action.

But if you run this script in Interaction Connect, execution continues immediately after the disposition request is sent. Because of this, the client status request is sent regardless of the outcome of the disposition request. This asynchronous behavior can lead to undesirable outcomes. Consider what happens if the initial disposition request fails and the client status request succeeds: the agent is "Available," but Dialer is waiting for them to send a disposition for the call on their queue.

To avoid these issues, you can add callbacks to each action and to each scripter object method. If you do this, the actions and object methods will be executed synchronously. Here's an improved version of the above sample code:

```java
{code:java}
function callComplete(disposition) {
    // Assign the disposition to the action.
    IS_Action_CallComplete.wrapupcode = disposition;


    // Assign a callback to be executed when the action completes.
    IS_Action_CallComplete.callback = function (error) {
        if (error) {
            /* The call disposition failed. Do not set the agent back to Available. Instead,
               log an error and take steps necessary to alert the agent to try again. */
            IS_Action_Trace.message = "The disposition failed.";
            IS_Action_Trace.level = 0;
            IS_Action_Trace.click();
        } else {
            // The call disposition was successful, set the agent back to Available.
            IS_Action_ClientStatus = "Available";
            IS_Action_ClientStatus.click();
        }
    }


    // Send the call disposition.
    IS_Action_CallComplete.click();
}
{code}
```

In this revised example, the callback function for the disposition request is only executed after the disposition has completed. If the disposition request is successful, no error is returned to the callback and the client status action is executed. This ensures that the agent does not become "Available" again until after the disposition completes successfully.

We've also added callbacks to each scripter object. Consider the following code snippet, which initializes a call object for the current dialer call, then attempts to disconnect the call:

```java
{code:java}
// Create a call object.
var currentDialerCall = scripter.createCallObject();


// Initialize the call object by assigning the current dialer call id to the call object id.
currentDialerCall.id = IS_Attr_CallId;


// Disconnect the call.
currentDialerCall.disconnect();
{code}
```

When the call object's id property is assigned, the client requests attributes for the specified interaction and initializes the call object once the appropriate information is returned. If this code is executed in Interaction Connect, the disconnect method can be executed before the call object has been initialized. You can prevent that from happening by using a new callback property (callObjectInitializedHandler) which allows execution to continue only after the call object has been initialized. Here's a better version of this functionality:

```java
{code:java}
// Create a call object.
var currentDialerCall = scripter.createCallObject();


// Assign a callback function to the callObjectInitializedHandler.
currentDialerCall.callObjectInitializedHandler = function () {
    currentDialerCall.disconnect();
}


// Initialize the call object by assigning the current dialer call id to the call object id.
currentDialerCall.id = IS_Attr_CallId;
{code}
```

By assigning a callback to the call object's callObjectInitializedHandler property, the revised example ensures that the call object's disconnect method is only called after this call object has been initialized.

Sample scripts

Source code is provided for the following sample scripts:
- Example 1 shows how to use IS_Action_QueryContactList and IS_Action_ManualOutboundCall actions to query a contact list and then populate a table of records within the custom script. This script allows the agent to click on records in the table to initiate a manual outbound dialer call.
- Example 2 shows how to initiate chats from a custom script using the IS_Action_PlaceChat action and how to send/receive chat messages using the chat object.

- Example 3 is a multi-page custom script that demonstrates some of the key actions and events you can use when writing a custom script in Interaction Scripter.

These scripts use callback functions for every action that needs to be executed in a specific order. Download the source code.

Example 1

This script uses IS_Action_QueryContactList and IS_Action_ManualOutboundCall actions to query the contact list and populate a table of records within the custom script. An agent can click on records in the table to initiate a manual outbound dialer call.
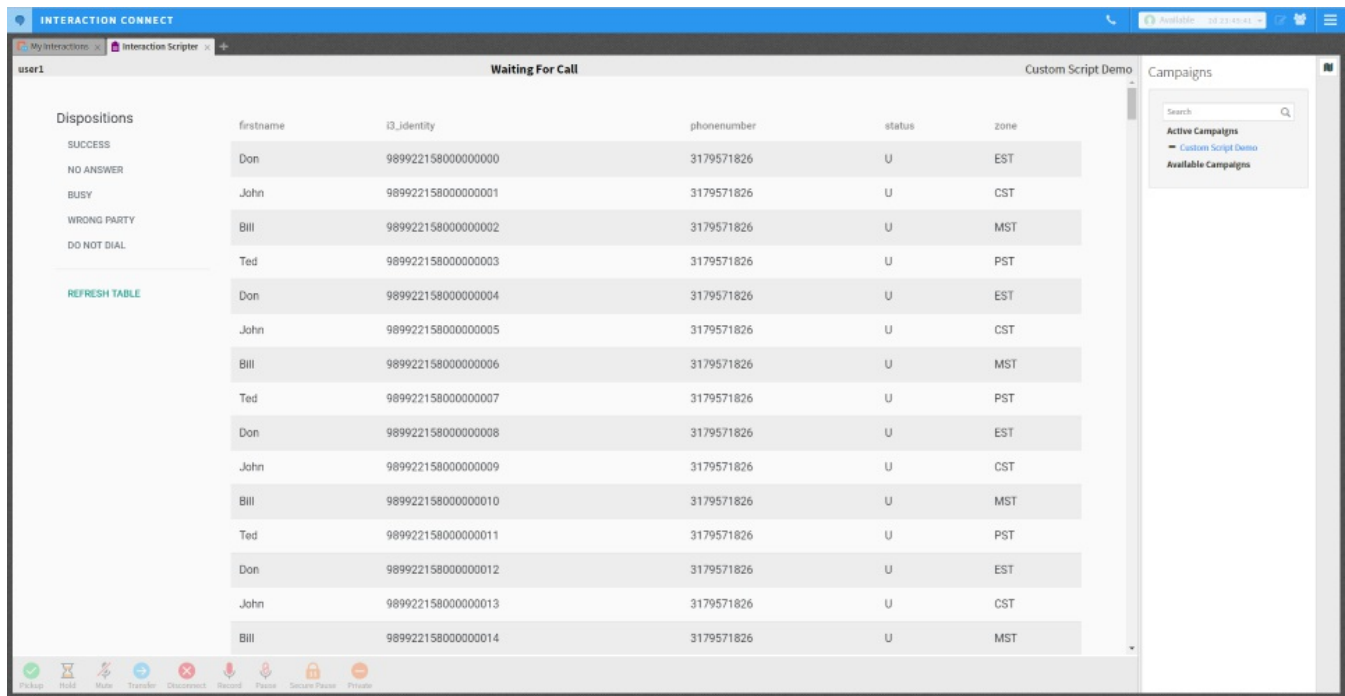
Example 1 source code

```html
<!doctype html>
<html lang="en">
<head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <!-- Actions -->
    <meta name="IS_Action_CallComplete">
    <meta name="IS_Action_QueryContactList">
    <meta name="IS_Action_ClientStatus">
    <meta name="IS_Action_ManualOutboundCall">
    <meta name="IS_Action_Disconnect">
    <meta name="IS_Action_Trace">
    <!-- Attributes -->
    <meta name="IS_Attr_CampaignId">
    <!-- Enable the command toolbar -->
    <meta name="IS_CommandToolbar_Visible" content="true">
    <!-- Bootstrap is used for layout and design -->
    <link rel="stylesheet" href="./bootstrap/fonts+icons.css">
    <link rel="stylesheet" href="./bootstrap/bootstrap-material-design.min.css">
</head>
<body>
    <div class="row m-5">
    <div class="col-md-2">
        <h5>Dispositions</h5>
        <button type="button" class="btn btn-block btn-secondary text-left m-0"
onclick="sendDisposition('Success')">Success</button>
        <button type="button" class="btn btn-block btn-secondary text-left m-0" onclick="sendDisposition('No
Answer')">No Answer</button>
        <button type="button" class="btn btn-block btn-secondary text-left m-0"
onclick="sendDisposition('Busy')">Busy</button>
        <button type="button" class="btn btn-block btn-secondary text-left m-0"
onclick="sendDisposition('Wrong Party')">Wrong Party</button>
        <button type="button" class="btn btn-block btn-secondary text-left m-0" onclick="sendDisposition('Do
Not Dial')">Do Not Dial</button>
        <hr class="featurette-divider"/>
        <button type="button" class="btn btn-block btn-primary text-left m-0"
onclick="queryContactList()">Refresh Table</button>
    </div>
    <div class="col-md-10">
```

18

```html
      <table class="table table-striped table-hover" id="contactRecordsTable"></table>
    </div>
    </div>
    <!-- Load jquery and popper first as it is required by Bootstrap. -->
    <script src="./bootstrap/jquery-3.2.1.slim.min.js"></script>
    <script src="./bootstrap/popper.js"></script>
    <script src="./bootstrap/bootstrap-material-design.js"></script>
    <script type="text/javascript">
        // When the DOM is ready, initialize Bootstrap and query the contact list.
        $(document).ready(function () { $('body').bootstrapMaterialDesign(); setTimeout(function () {
queryContactList(); }); });
        // The queryContactList function is called after the DOM is loaded initially,
        // or when the "Refresh Table" button is clicked.
        function queryContactList() {
            IS_Action_QueryContactList.displayName = "Example Table";
            IS_Action_QueryContactList.tableName = "EXAMPLETABLE";
            IS_Action_QueryContactList.connectionId = "{F028F7C3-089A-48D8-9DCC-C1A4CA53FB5D}";
            IS_Action_QueryContactList.statement = "select i3_identity, firstname, lastname, status,
phonenumber from EXAMPLETABLE";
            /* Assign a callback which will be invoked after the action has been exectued.
            The response contains an error message indicating whether or not the query was
            successful. */
            IS_Action_QueryContactList.callback = function (response) {
                if (response.errorMessage === "") {
                    // The query was successful. Process the returned records and update
                    // the contact list table.
                    loadContactTable(response.records);
                } else {
                    // The query failed. Log an error message to the console.
                    IS_Action_Trace.message = "Failed to query the contact list.";
                    IS_Action_Trace.level = 0;
                    IS_Action_Trace.click();
                }
            }
            // Execute the action.
            IS_Action_QueryContactList.click();
        }
        /* The loadContactTable function builds an HTML string which is used to populate
        the returned records into the contact list table. The function is called whenever
        the IS_Action_QueryContactList action is successful. */
        function loadContactTable(records) {
            // Start by generating HTML for the table's headers.
            var contactTableHTML = "<thead><tr>";
            var tableHeaders = Object.keys(records[0].values);
            for (var i = 0; i < tableHeaders.length; i++) {
                contactTableHTML += "<th>" + tableHeaders[i] + "</th>";
            }
            contactTableHTML += "</tr></thead>";
            /* Next, loop through each record and generate HTML for the table body.
            Each record returned by the query will have its own row in the table.
            If a row is clicked, a manual outbound call is attempted on the corresponding
            record. */
            contactTableHTML += "<tbody style='cursor: pointer;'>";
            for (var i = 0; i < records.length; i++) {
                var contactIdentity = records[i].identity;
                var contactPhoneNumber = records[i].values.phonenumber;
                contactTableHTML += "<tr onclick='placeManualOutboundCall(\"" + contactIdentity + "\" , \"" +
contactPhoneNumber + "\")'>";
                var tableValues = Object.values(records[i].values);
                for (var j = 0; j < tableValues.length; j++) {
                    contactTableHTML += "<td>" + tableValues[j] + "</td>";
                }
                contactTableHTML += "</tr>";
            }
            contactTableHTML += "</tbody>";
            // Populate the table with the generated HTML.
            document.getElementById("contactRecordsTable").innerHTML = contactTableHTML;
        }
        // The placeManualOtuboundCall function is called whenever a row in the contact
        // list table is clicked.
```

```
        function placeManualOutboundCall(identity, phoneNumber) {
            IS_Action_ManualOutboundCall.i3identity = identity;
            IS_Action_ManualOutboundCall.campaignid = "25A69636-DEE1-40E3-8FAF-EC92F0A6384F";
            IS_Action_ManualOutboundCall.contactcolumnid = "1";
            IS_Action_ManualOutboundCall.phonenumber = phoneNumber;
            // Initiate the manual outbound call.
            IS_Action_ManualOutboundCall.click();
        }
        function sendDisposition(disposition) {
            // Disconnect the call.
            IS_Action_Disconnect.click();
            IS_Action_CallComplete.wrapupcode = disposition;
            // Assign a callback to be invoked after the IS_Action_CallComplete action
            // has been executed.
            IS_Action_CallComplete.callback = function (error) {
                if (error) {
                    // The IS_Action_CallComplete action failed, log an error.
                    IS_Action_Trace.message = "The disposition failed.";
                    IS_Action_Trace.level = 0;
                    IS_Action_Trace.click();
                } else {
                    // The disposition was successful. Set the agent back to available.
                    IS_Action_ClientStatus.statuskey = "Available";
                    IS_Action_ClientStatus.click();
                }
            }
            // Execute the action.
            IS_Action_CallComplete.click();
        }
    </script>
</body>
</html>
```

Example 2

This script initiates a chat using the IS_Action_PlaceChat action. It uses a chat object to send and receive chat messages.
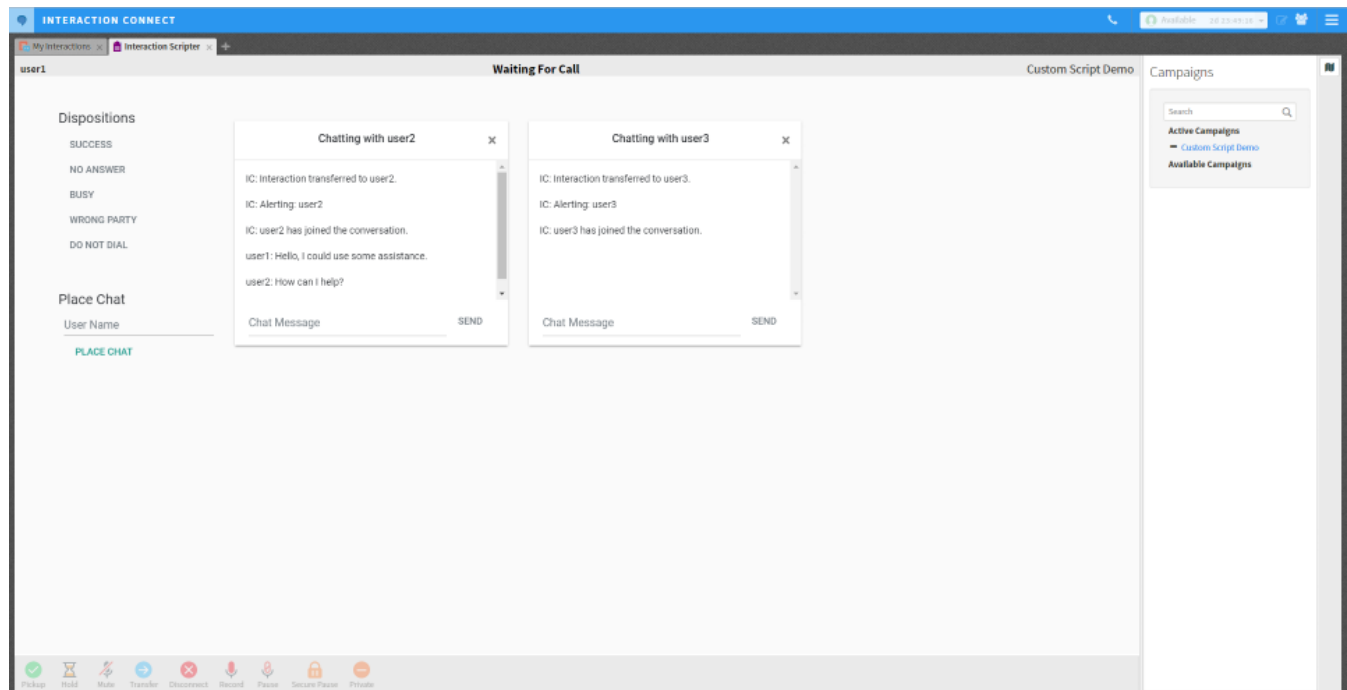
Example 2 source code

```html
<!doctype html>
<html lang="en">
<head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <!-- Actions -->
    <meta name="IS_Action_CallComplete">
    <meta name="IS_Action_ClientStatus">
    <meta name="IS_Action_PlaceChat">
    <meta name="IS_Action_Disconnect">
    <meta name="IS_Action_Trace">
    <!-- Enable the command toolbar -->
    <meta name="IS_CommandToolbar_Visible" content="true">
    <!-- Bootstrap is used for layout and design -->
    <link rel="stylesheet" href="./bootstrap/fonts+icons.css">
    <link rel="stylesheet" href="./bootstrap/bootstrap-material-design.min.css">
</head>
<body>
    <div class="row m-5">
        <div class="col-md-2">
            <h5>Dispositions</h5>
            <button type="button" class="btn btn-block btn-secondary text-left m-0"
onclick="sendDisposition('Success')">Success</button>
            <button type="button" class="btn btn-block btn-secondary text-left m-0"
onclick="sendDisposition('No Answer')">No Answer</button>
            <button type="button" class="btn btn-block btn-secondary text-left m-0"
onclick="sendDisposition('Busy')">Busy</button>
            <button type="button" class="btn btn-block btn-secondary text-left m-0"
onclick="sendDisposition('Wrong Party')">Wrong Party</button>
            <button type="button" class="btn btn-block btn-secondary text-left m-0"
onclick="sendDisposition('Do Not Dial')">Do Not Dial</button>
            <h5 class="mt-5">Place Chat</h5>
            <div class="pl-2">
                <input type="text" class="form-control mb-1" placeholder="User Name" id="userName" />
                <button type="button" class="btn btn-primary btn-block text-left" onclick="placeChat()">Place
Chat</button>
            </div>
        </div>
        <div class="col-md-10">
            <div class="row" id="chatArea">
            </div>
        </div>
    </div>
    <!-- Load jquery and popper first as it is required by Bootstrap. -->
    <script src="./bootstrap/jquery-3.2.1.slim.min.js"></script>
    <script src="./bootstrap/popper.js"></script>
    <script src="./bootstrap/bootstrap-material-design.js"></script>
    <script type="text/javascript">
        // When the DOM is ready, initialize Bootstrap
        $(document).ready(function () { $('body').bootstrapMaterialDesign(); });
        // Keep track of the agent's chats
        var currentChats = [];
        function placeChat() {
            // Get the specified recipient from the input box
            IS_Action_PlaceChat.recipient = document.getElementById("userName").value;
            IS_Action_PlaceChat.callback = function(interactionId) {
                // Create a chat object and add it to the user's current chats
                currentChats[interactionId] = scripter.createChatObject();
                // Use the chatObjectInitializedHandler to assign a callback to be executed
                // whenever the chat object's messages have changed.
                currentChats[interactionId].chatObjectInitializedHandler = function(chatObject) {
                    currentChats[interactionId].SubObjectChangeHandler = chatMessagesChanged;
                    // Open a chat box within the script.
                    renderChatBox(currentChats[chatObject.id]);
                }
                currentChats[interactionId].id = interactionId;
            }
            IS_Action_PlaceChat.click();
        }
```

```javascript
        function renderChatBox(chatObject) {
            // Build HTML for a chat box using a Bootstrap card and card-deck.
            var chatBoxHTML = "<div class='col-md-4 mt-3' id='" + chatObject.id + "'>";
            chatBoxHTML += "<div class='card-deck'>";
            chatBoxHTML += "<div class='card text-center'>";
            // Add a card header to indicate who the agent is chatting with.
            chatBoxHTML += "<div class='card-header'>";
            chatBoxHTML += "<h6 class='d-inline'>Chatting with " + chatObject.remoteName + "</h6>";
            chatBoxHTML += "<button type='button' class='close' onclick='closeChat(" + chatObject.id + ")'
aria-label='Close'><span aria-hidden='true'>&times;</span></button>";
            chatBoxHTML += "</div>";
            // Add a body to the card which will contain the chat messages.
            chatBoxHTML += "<div class='card-body chatMessages' id='" + chatObject.id + "' style='overflow-y:
scroll; height: 200px; padding: 1rem;'></div>";
            // Add a footer to the card with an input box and send button for the agent to
            // send messages.
            chatBoxHTML += "<div class='card-footer text-muted' style='border-top-style: none'>";
            chatBoxHTML += "<div class='d-flex'>";
            chatBoxHTML += "<input type='text' class='form-control chatMessage d-inline' placeholder='Chat
Message' id='" + chatObject.id + "'>";
            chatBoxHTML += "<button class='btn btn-secondary' type='button' onclick='sendChat(" +
chatObject.id + ")'>Send</button>";
            chatBoxHTML += "</div></div>";
            // Closing divs for the column, card-deck, and card.
            chatBoxHTML += "</div></div></div>";
            // Add the chat box html to the chat area in the script.
            $("#chatArea").append(chatBoxHTML);
        }
        // The chatMessagesChanged function is assigned to each chat object's
        // SubObjectChanged handler and is called whenever the chat object has changed.
        function chatMessagesChanged(messages) {
            if (messages.length > 0) {
                var chatId = messages[0].chatMember.interactionId;
                var messageHTML = "";
                // Build HTML for the chat message.
                messageHTML += "<p class='text-left'>" + messages[0].chatMember.displayName + ": " +
messages[0].text + "</p>";
                // Append the chat message to the chat box associated with the chat.
                $("#" + chatId + ".chatMessages").append(messageHTML);
            }
        }
        function closeChat(interactionId) {
            currentChats[interactionId].disconnect();
            // Remove the chat box from the script.
            $("#" + interactionId).remove();
        }
        function sendChat(interactionId) {
            alert("Sending Chat!");
            // Get the message from the associated chat box
            var message = $("#" + interactionId + ".chatMessage").val();
            // Send the message from the associated chat object
            currentChats[interactionId].sendChatMessage(message);
        }
        function sendDisposition(disposition) {
            // Disconnect the call.
            IS_Action_Disconnect.click();
            IS_Action_CallComplete.wrapupcode = disposition;
            // Assign a callback to be invoked after the IS_Action_CallComplete action has
            // been executed.
            IS_Action_CallComplete.callback = function (error) {
                if (error) {
                    // The IS_Action_CallComplete action failed, log an error.
                    IS_Action_Trace.message = "The disposition failed.";
                    IS_Action_Trace.level = 0;
                    IS_Action_Trace.click();
                } else {
                    // The disposition was successful. Set the agent back to available.
                    IS_Action_ClientStatus.statuskey = "Available";
                    IS_Action_ClientStatus.click();
                }
```

```
        }
        // Execute the action.
        IS_Action_CallComplete.click();
    }
    </script>
</body>
</html>
```

Example 3

The Hawaiian Cruises script is a multi-page custom script that demonstrates some of the key actions and events you can use when writing a custom script in Interaction Scripter. This script works with power, predictive and preview campaigns and allows an agent to:

- Go available
- Receive and view data pops
- Update contact attributes
- Dispose of dialer interactions
- Skip preview calls
- Request breaks
- Schedule callbacks

This script is a modified version of the Hawaiian Cruises script used in Interaction Connect. Because it has all of the functionality of the original script, comparing the two versions can help you understand what you need to do differently when developing custom scripts for the two products.

Example 3 source code

Download the source code. See Also Writing custom scripts for Interaction Connect or Scripter .NET Sample ICWS Dialer Web Application
Last updated 2018-03-26 14:28:02 EDT

Sample ICWS Dialer Web Application

A sample IC web services application is provided to show how common campaign tasks can be coded. Before studying the source code to learn how the application is constructed, here's how to run the app:

Download the example application

1. Click here to download the application source code (icws-dialer.zip).
2. Extract contents of **ic-ws-dialer.zip** file to your hard drive.

Launch the app and connect to ODS

1. Navigate to to the location where you extracted the zip. Open **index.html** in a web browser.
2. Enter credentials required to access your Outbound Dialer Server:

User Name

     CIC user name.

Password

     Password for CIC user account.

Station

     CIC station name.

Server

     Name of the Outbound Dialing Server.

3. Click **Sign in**.

Interact with Calls

The figure below shows a server running "Test Campaign" in Preview mode. A preview screen pop displays information about the contact. The attributes of the call were made visible by clicking **Attributes**.

PREVIEWING CALL

MY STATUS

Campaign Call ▼

MY ACTIONS

Place Preview Call

Skip Preview Call

Disconnect Call

Request Dialer Break

DISPOSITIONS

Success

No Answer

Wrong Party

Busy

ACTIVE CAMPAIGNS

Test Campaign

AVAILABLE CAMPAIGNS

Data Pop Info
__type: urn:inin.com:dialer:previewPop
interactionId: 2001760011
campaignId: 399DF4B9-F765-4662-B1CC-6545287F533C
dialingMode: 0
dispositions: [object Object],[object Object],[object Object],[object Object],[object Object], [object Object],[object Object],[object Object],[object Object],[object Object]
autoDisconnectOnDisposition: true
allowAgentCallbacks: true
allowSkip: true
callPlaced: false
Attributes
is_attr_I3_ATTEMPTSRESCHEDULED: 0
is_attr_I3_IDENTITY: 215537247000000004
is_attr_I3_UPLOAD_ID: 136
is_attr_STATUS: C
is_attr_badnumber: 5411230005
is_attr_callid: 2001760011
is_attr_campaign_type: 3
is_attr_campaignid: {399DF4B9-F765-4662-B1CC-6545287F533C}
is_attr_campaignname: Test Campaign
is_attr_contactcolumnname: PHONE
is_attr_dialingmode: 1
is_attr_firstName: Karen
is_attr_lastName: Puckett
is_attr_middleInitial: U
is_attr_numbertodial: 3175557637
is_attr_phone: 3175557637
is_attr_sex: F
is_attr_skill: 2
is_attr_work: 5414270005

1. To dial the contact, click **Place Preview Call**.
   - If you click **Skip Preview Call**, a different contact will be displayed from the contact list.
   - If you click **Request Dialer Break**, a break will be granted after the current call ends. You must select a disposition to conclude the current call.

     Your break begins when the current call ends. To end a break, click **End Dialer Break**.
2. The banner displays **ON DIALER CALL** to indicate that the call connected to the party.

Log Out

ON DIALER CALL

MY STATUS

Campaign Call ▾

MY ACTIONS

Place Preview Call

Skip Preview Call

Disconnect Call

Pending Dialer Break...

DISPOSITIONS

Success

No Answer

Wrong Party

Busy

ACTIVE CAMPAIGNS

Test Campaign

AVAILABLE CAMPAIGNS

Data Pop Info
__type: urn:inin.com:dialer:dataPop
interactionId: 2001760011
campaignId: 399DF4B9-F765-4662-B1CC-6545287F533C
dialingMode: 0
dispositions: [object Object],[object Object],[object Object],[object Object],[object Object],
[object Object],[object Object],[object Object],[object Object],[object Object]
autoDisconnectOnDisposition: true
allowAgentCallbacks: true
Attributes
is_attr_I3_ATTEMPTSRESCHEDULED: 0
is_attr_I3_IDENTITY: 215537247000000004
is_attr_I3_UPLOAD_ID: 136
is_attr_STATUS: C
is_attr_badnumber: 5411230005
is_attr_callid: 2001760011
is_attr_campaign_type: 3
is_attr_campaignid: {399DF4B9-F765-4662-B1CC-6545287F533C}
is_attr_campaignname: Test Campaign
is_attr_contactcolumnname: PHONE
is_attr_dialingmode: 1
is_attr_firstName: Karen
is_attr_lastName: Puckett
is_attr_middleInitial: U
is_attr_numbertodial: 3175557637
is_attr_phone: 3175557637
is_attr_sex: F
is_attr_skill: 2
is_attr_work: 5414270005

3. Select a disposition to conclude the call. Click **Success**, **No Answer**, **Wrong Party**, or **Busy**.

Log Out

You must conclude the current call before logging out. Disposition the call and then click **Log Out**.

See also

Sample Interaction Connect scripts

Capitalization conventions

When writing scripts, comply with the following capitalization conventions:

1. **Use upper camel case for class names**. Capitalize each word of a class name. Examples are `CallObject`, `ChatObject`, and `QueueObject`.
2. **Use lower camel case for method names and variables**. In lower camel case, the first word is lower case and all subsequent words are capitalized. Examples are:

   `CallObject.dial`

   `var objCall = Scripter.createCallObject();.`
3. **Use lower case for attributes and properties**. Examples are:

   callObject1.dial(remotenumber, false);

   IS_Action_Trace.message = "This is a test.";

   > **NOTE:** When a script is executes, attribute names in IS_Actions are automatically converted to lower case. It is not necessary to update existing scripts to make those attributes lower case.

IceLib.Dialer API Documentation

IceLib.Dialer is an API for creating custom dialing clients and applications that configure Interaction Dialer. Related documentation is hosted on the PureConnect Developer Portal at https://help.genesys.com/developer/cic/docs/dialericelib/webhelp/index.html.

Interaction Scripter Debugger

Interaction Scripter offers a debugging feature that helps developers detect and resolve problems with custom campaign scripts. The debugger is available only when Interaction Scripter is started with the Debug command-line parameter.

When you run Scripter in Debug mode, it analyzes events in the current session to identify potential problems with scripts. When a problem is detected, Scripter sends an error message to the debugger. In most cases you must manually access the Debug Dialog window to view the error messages . However, the Debug Dialog window automatically opens when scenarios like the following are detected:

- Agent sets status to Available without dispositioning the current campaign call.
- Agent received a datapop while a disposition is pending.
- A page unload event was called, but a corresponding onload event did not occur within 5 seconds.
- Multiple page elements are assigned the same 'name' attribute.
- Scripter detects COM errors related to invalid CallIDs, conference ids, etc.
- A SetAvailability call is made while an Agent is on a call or is in 'Awaiting Callback' status.
- Client status changes to an Available state while the Agent is processing a call.
- A call has not been dispositioned prior to transfer.
- A logout request occurs while in "Awaiting Callback" status.
- An Agent's Workgroup activation changes unexpectedly.

Launch Scripter in Debug mode

To launch Scripter in Debug mode, you add the \debug parameter to the Interaction Scripter command line:

InteractionScripter.NET.exe /debug

When you do, a Debug menu appears in the Interaction Scripter window.
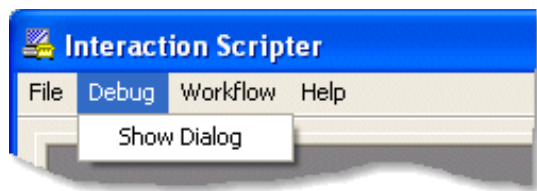
From the menu select the Show Dialog command and you'll see the Debug Dialog window. Error messages are displayed in a list at the top of the Debug Dialog. Columns in the list indicate the time, Script Page filename, and text of each error. The dialog also displays the URL of the selected script page. Tab pages on the dialog display the full text of the error message and may offer suggestions for resolving the error, script snippets that show how to resolve error, and contextual information about the scenario.

Time when debug
message was
generated

HTML page that
contains script
error

Description of
the problem

Debug Dialog

Time	Script Page	Error Message
08:48:43	index.html	An element with the name 'IS_Attr_Cal...
08:48:43	index.html	An element with the name 'IS_Attr_Cal...
08:48:43	index.html	Action 'ClientStatus' uses deprecated '...
08:48:45	introduction.htm	An element with the name 'IS_Attr_AC...
08:48:45	introduction.htm	An element with the name 'IS_Attr_AC...
08:48:45	introduction.htm	An element with the name 'IS_Attr_PR...
08:48:45	introduction.htm	An element with the name 'IS_Attr_PR...
08:48:45	introduction.htm	An element with the name 'IS_Attr_PR...
08:48:45	introduction.htm	An element with the name 'IS_Attr_PR...

Error Message list

URL of page selected
in Error Message list

Icon indicates
that the tab
contains
information
about the error

Script URL: file://dialerdb/f/scripts/rant/index.html

Course of action
to resolve error

Contents of Stack

Example code,
(where applicable)

ⓘ Error | ⓘ Suggestion | Example | Stack

An element with the name 'IS_Attr_CallId' already exists.

Close   Clear   Help

Close Debug
Dialog

Clear messages
logged since Scripter
was started

Open online
help topic

Controls on the Debug Dialog

Error Message list box

Messages are displayed in a list at the top of the *Debug Dialog*. Columns in the list indicate the time, Script Page filename, and text of each error.

Script URL Field

This read-only field displays the URL of the selected script page.

Close button

This button closes the Debug Dialog. If you reopen it later, messages logged since the current Scripter session started will be displayed in the list, unless messages were previously cleared.

Clear button

Removes all messages logged since Scripter was started.

Tab pages

The tab pages at the bottom of the dialog display the full text of the selected error message, and may optionally offer suggestions for resolving the error, example code, and other information. The ⓘ icon appears on tab pages that are not empty.

Error

This tab displays the text of the selected error message, which may be too long to view in the message list.

Suggestion

This tab may contain suggestions for resolving the error.

Example

This tab may contain a script snippet that shows how to resolve the error. (Most messages do not have corresponding code samples.) Text in this window can be copied to the clipboard. To copy text, make a selection, press the right mouse button, and then choose *Copy* from the context menu.

Stack

This tab may provide contextual information regarding a situation. For example, if the script attempts to disconnect, when there is no active call object, that context would be described on the stack page.

Interaction Scripter Actions

Actions are messages from Interaction Scripter scripts to the server that trigger an action on the CIC server. This section explains the various actions that are available in Scripter and how to implement code in a web pages to use these actions.

Standard Actions

Standard Actions perform normal operations on phone calls, such as picking up a call, placing it on hold, transferring, or recording. Standard actions provide basic telephony integration with the PureConnect platform. These actions allow scripts to manipulate telephone calls.

Predictive Actions

Predictive Actions are valid if Interaction Dialer is installed on the CIC Server and the user is logged into Dialer. Predictive actions apply to Interaction Dialer campaigns and campaign calls. Predictive actions are also useful in preview mode, when information about a party is pushed to an agent before the agent initiates the call.

Using Scripter actions is a two step process

1. First the action must be defined in the web page. The action is defined in an HTML meta tag, this meta tag is then parsed by Scripter when the page is loaded. This allows Scripter to set up specific event handlers to handle when the action is called from within the body of the loaded web page. If the action is not defined in a meta tag, Scripter will not know how to respond to the action, and an error will be received in Scripter. If in debug mode an entry will also be made in the debug window indicating that the action is not defined.   The meta tag definition of the action and the calling action itself much match in case. The meta tag definition is case sensitive.
2. The second step in setting up an action is to use the action in a function definition within JavaScript. The function definition is usually in response to an event handler in the web page, like handling a button click event or a drop down value change event for example. When calling the Scripter action, the calling action is normally the name of the action followed by a .click().

> **NOTE:** A few actions should be used with care, since they potentially affect the performance of a script or server. Actions having scale impact include:
> - IS_Action_QueryContactList—an inefficient or overly broad query could affect the performance of a database server.
> - Dialer.sendCustomHandlerlNotification Method—starting a very complex, long-running handler could affect the performance of a PureConnect server.

Standard Actions

The Interaction Scripter standard actions can be used in an inbound script or an outbound script. These actions are loaded into the base view in Interaction Scripter. These actions do not require the user to be logged into Dialer, thus can be used in a web page that is auto loaded within Scripter for handling inbound calls in a blended environment.

Generally speaking, standard actions are operations that scripts can perform on call objects. Standard actions emulate the functionality of the CIC client. Most standard actions manipulate a telephone call. There are standard actions that pick up, listen to, mute, disconnect, record, and transfer calls, for example. To indicate which call to process, CallID can be specified as an attribute. If a CallID is not provided, the current or active call of the current campaign in the queue is assumed to be the target of the action.

Additional standard actions can change an agent's client status, tab page, toggle full screen mode, or make the Scripter window topmost. The table below lists standard actions and attributes. Optional attributes are enclosed in brackets. Interaction Scripter standard actions can be used in an inbound script or an outbound script. These actions are loaded into the base view in Interaction Scripter.

Standard actions do not require the user to be logged into Dialer, and therefore can be used in a web page that is auto loaded within Scripter for handling inbound calls in a blended environment.

In this section, each action is discussed and a code sample for each action is provided.

Action	Definition
IS_Action Close	Closes the current tab page.
IS_Action_ClientStatus	Sets agent status.
IS_Action_CompleteConsult	This action ends a consult call in scripts for Interaction Connect only.
IS_Action_Disconnect	Disconnect a call
IS_Action_Exit	Terminates Interaction Scripter Client.
IS_Action_Hold	Put a call on hold.
IS_Action_Listen	Listen to a call.
IS_Action_Mute	Mute a call.
IS_Action_Park	Park a call.
IS_Action_Pickup	Pick up a call.
IS_Action_PlaceCall	Place a call.
IS_Action_PlaceChat	Initializes a chat between the user and another user.
IS_Action_PlayWav	Play pre-recorded message to the call.
IS_Action_Private	Prevent others from listening to call.
IS_Action_Record	Record a call.
IS_Action_RecordPause	Pause a recording.
IS_Action_SelectPage	Sets focus to the current tab page.
IS_Action_SendToVoiceMail	Send a call to voice mail.
IS_Action_SetForeground	Brings application window to the top.
IS_Action_Trace	Creates a trace log entry.
IS_Action_Transfer	Transfer a call.

IS_Action_ClientStatus

Definition

This action provides the ability to change a user status within a script. It changes the agent's CIC client *status*—an availability indicator that affects the processing of calls directed to the agent. To receive calls in a campaign, an agent's client status must be set to an "available" status condition code. Status indicators are defined in Interaction Administrator's "Status Messages" container (as Message Names). Some of the default status indicators are:

At a Training Session	At Lunch	Available
Available, Follow-Me	Available, Forward	Available, No ACD
Away From desk	Do Not Disturb	Gone Home
In a Meeting	On Vacation	Out of the Office
Out of Town	Working at Home	

Any status other than "Available", "Available, Forward" or "Available, No ACD" sends incoming calls to the agent's voice mailbox. For example, an agent whose status is "At Lunch" will not receive calls.

Attributes

This action as three properties, only one is required the other two are optional. The code snippet below uses statuskey, which is the localized value of the status. This is appropriate to use if the script is being developed for another language. The IS_Action_ClientStatus element accepts the following attributes:

statusid

> The value to change the users' status to (e.g. "Available").

[until]

> Optional DATETIME that determines when this status condition will expire. Think of this as the date and time that the status is valid until. This only applies to statuses that allow DateTime option.

[statuskey]

> Optional language-independent attribute that allows a script to change status by specifying a message name, such as "Available", rather than a localized status Message. This frees the script developer from having to know the localized version of status strings.
>
> For example, a script might set IS_Action_ClientStatus.statuskey = "AcdAgentNotAnswering", rather than use IS_Action_ClientStatus.statusid = "ACD – Agent Not Answering", which is language-dependent. For more information, see the *Status Messages* container in Interaction Administrator.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_ClientStatus() {
  IS_Action_ClientStatus.statusid = 'Available';
  IS_Action_ClientStatus.callback = function(error) {
    if (error) {
      console.error("IS_Action_ClientStatus failed.");
    } else {
      console.log("IS_Action_ClientStatus succeeded.");
    }
  }
}
```

Example 1

Example 1 paints a button that sets the client status to the "Available".

```html
<head>
    <script language=javascript>
        function IS_Event_PreviewTimeoutStopped(args) {
            var id = args.interactionId;
            // insert other code here as needed...
        }
    </script>
</head>
```

Example 2

This status button example invokes a user-defined script function named "SetClientStatus" to change the agent's client status to "Available". When the button is clicked, the actual element that fires the event is a <meta> element in the non-visible <head> section of the document. The <meta> element in the <head> section of the HTML page instantiates the action as a non-visual object.

```html
<head>
    <meta name="IS_Action_ClientStatus">
    <script language=javascript>
        function SetClientStatus(StatusId) {
            IS_Action_ClientStatus.statusid = StatusId;
            IS_Action_ClientStatus.click();
        }
    </script>
</head>
<body>
  <input type=button value="Available" onclick="SetClientStatus('Available');"
</body>
```

Example 3

```html
<head>
    <meta name="IS_Action_ClientStatus">
    <script language=JavaScript>
        function IS_ChangeUserStatus(p_statusString, p_dtUntilDateTime) {
            IS_Action_ClientStatus.statuskey = p_statusString;
            if (p_dtUntilDateTime != null) {
                IS_Action_ClientStatus.until = p_dtUntilDateTime;
            }
            IS_Action_ClientStatus.click();
        }
    </script>
</head>
<body>
    <input type=button value="Available" onclick="IS_ChangeUserStatus('Available');">
</body>
```

IS_Action_CompleteConsult

Definition

This action ends a consult call in scripts for Interaction Connect only.

When a consult transfer action is called from a custom script in Scripter.NET, an IceLib call is made to complete the consult interaction. The consulted party and the original party are connected and the agent is removed from the call.

But in Interaction Connect, an Interaction Center Web Services (ICWS) call must be made to start a consult interaction before the consult transfer can be completed. Because of this difference in client behavior, developers should use the "IS_Action_CompleteConsult" action to cancel or complete a consult transfer.

Attributes

action

> Set this attribute to "complete" to complete the consult call, or "cancel" to cancel the consult call.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_CompleteConsult() {
  IS_Action_CompleteConsult.action = "complete";
  IS_Action_CompleteConsult.callback = function(error) {
    if (error) {
      console.error("IS_Action_CompleteConsult failed.");
    } else {
      console.log("IS_Action_CompleteConsult succeeded.");
    }
  }
}
```

Example

To create a consult transfer:

```
IS_Action_Transfer.consult = true;
IS_Action_Transfer.recipient = "1234";
IS_Action_Transfer.click(); // This will initiate the call to "1234".
// The dialer call is placed on hold and the agent is speaking with the consulted party.
```

At this point, the "audience" attribute of an IS_Action_Transfer can be used to toggle between participants of the consult interaction.

To complete the consult transfer:

```
IS_Action_CompleteConsult.action = "complete";
IS_Action_CompleteConsult.click();
```

To cancel the consult transfer:

```
IS_Action_CompleteConsult.action = "cancel";
IS_Action_CompleteConsult.click();
```

IS_Action_Close

Definition

Each campaign script is displayed on its own tab page in Interaction Scripter. IS_Action_Close closes the current tab page. If on a Dialer call, close is only performed once the current call is disconnected/transferred and dispositioned. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Close() {
  IS_Action_Close.callback = function(error) {
    if (error) {
      console.error("IS_Action_Close failed.");
    } else {
      console.log("IS_Action_Close succeeded.");
    }
  }
}
```

Example 1

This example creates a button that closes the current tab page

```
<body>
    <input type=button name="IS_Action_Close" value="Close">
</body>
```

Example 2

This status button example invokes a user-defined script function named "Close" to close the current tab page. When the button is clicked, the actual element that fires the event is a <meta> element in the non-visible <head> section of the document. The <meta> element in the <head> section of the HTML page instantiates the action as a non-visual object.

```
<head>
    <meta name="IS_Action_Close">
    <script language="JavaScript">
        function IS_CloseTab() {
            IS_Action_Close.click();
        }
    </script>
</head>
<body>
    <input type=button value="Close Tab" onclick="IS_CloseTab();">
</body>
```

IS_Action_Disconnect

Definition

This call control action provides the ability to disconnect either the current call that is on the current users queue, or by passing in a valid `callid` of a call, it will disconnect that call. IS_Action_Disconnect does not generate an error if the party hangs up before this action is called. This action disconnects an agent from the call specified by the `callid` attribute. The current or active call of the current campaign in the queue is assumed to be the target of this action if the `callid` attribute is not set. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Disconnect element accepts `callid` as an optional attribute:

[callid]

> Optional. The `callid` to which this action applies (e.g. "89900001"). Dialer scripts use the current Dialer `callid` by default.A "debug mode" error is logged for non-Dialer scripts if the `callid` attribute is not specified.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Disconnect() {
  IS_Action_Disconnect.callback = function(error) {
    if (error) {
      console.error("IS_Action_Disconnect failed.");
    } else {
      console.log("IS_Action_Disconnect succeeded.");
    }
  }
}
```

Example 1

This example paints a "Disconnect" button that allows the agent to disconnect a call.

```
<body>
    <input type=button name="IS_Action_Disconnect" value="Disconnect">
</body>
```

Example 2

The "Disconnect" button in this example invokes a user-defined "Disconnect" script function to disconnect a call. When the button is clicked, the actual element that fires the event is a <meta> element in the non-visible <head> section of the document. The <meta> element in the <head> section of the HTML page instantiates the action as a non-visual object.

```
<head>
    <meta name="IS_Action_Disconnect">
    <script language=javascript>
        function Disconnect() {
            IS_Action_Disconnect.click();
        }
    </script>
</head>
<body>
    <input type=button value="Disconnect" onclick='Disconnect();'>
</body>
```

Example 3

This example passes the callid of the call to disconnect.

```
<head>
    <meta name="IS_Action_Disconnect">
    <script language="javascript">
        function IS_DisconnectCall(p_page, p_CallId) {
            if (p_CallId != null) {
                IS_Action_Disconnect.callid = p_CallId;
            }
            IS_Action_Disconnect.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Disconnect Call" onclick="IS_DisconnectCall(null,IS_ATTR_CallId.value);">
</body>
```

IS_Action_Exit

Definition

This action provides the ability to exit Scripter from within the web page that is loaded in Scripter. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete").If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Exit element accepts the following attributes:

[endtask]

> This Boolean attribute is optional. When set to false, the script will request logouts from Dialer and wait for those request to be granted. Once the Dialer logout requests have been granted, the script will exit both the Dialer session and Scripter. When set to true, which is the default*, the script will immediately close Scripter. The Dialer session will continue running until a timeout occurs. If you don't specify the EndTask attribute, the result will be the same as setting the attribute to true.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Exit() {
  IS_Action_Exit.endtask=false;  IS_Action_Exit.callback = function(error) {
    if (error) {
      console.error("IS_Action_Exit failed.");
    } else {
      console.log("IS_Action_Exit succeeded.");
    }
  }
}
```

Example 1

This example paints a button that closes Interaction Scripter.

```
<body>
    <input type=button name="IS_Action_Exit" value="Exit">
</body>
```

Example 2

```
function Exit() {
    IS_Action_Exit.endtask = false;
    IS_Action_Exit.click();
}
```

Example 3

```
function Exit() {
    IS_Action_Exit.click();
}
```

IS_Action_Hold

Definition

This action places the current call (or the call specified by the `callid` attribute) on hold. To take a call off hold, an agent must invoke the IS_Action_Pickup element. The current or active call of the current campaign in the queue is assumed to be the target of this action, if the `callid` attribute is not set. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Hold element has the following attribute:

[callid]

> The `callid` to which this action applies (e.g. "89900001"). The `callid` is optional. Dialer scripts use the current Dialer `callid` by default. A "debug mode" error is logged for non-Dialer scripts if `callid` attribute is not specified.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all [Interaction Scripter actions](#) (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See [Writing custom scripts for Interaction Connect or Scripter .NET](#).

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Hold() {
  IS_Action_Hold.callback = function(error) {
    if (error) {
      console.error("IS_Action_Hold failed.");
    } else {
      console.log("IS_Action_Hold succeeded.");
    }
  }
}
```

Example 1

This is an example of a "Hold" button that allows the agent to hold a call.

```
<body>
    <input type=button name="IS_Action_Hold" value="Hold">
    <input type=button name="IS_Action_Pickup" value="Pickup">
</body>
```

Example 2

This example places the call specified by the `callid` attribute on hold.

```
<head>
    <meta name="IS_Action_Hold">
    <script language="javascript">
        function IS_HoldCall(p_page, p_CallId) {
            if (p_CallId != null) {
                IS_Action_Hold.callid = p_CallId;
            }
            IS_Action_Hold.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Put Call On Hold" onclick="IS_HoldCall();">
</body>
```

Example 3

This is an example of a toggle-style "Hold" button that invokes the "Hold" script function. Note that <meta> elements in the <head> section of the HTML& document instantiate the actions as non-visual objects.

```html
<head>
    <meta name="IS_Action_Hold">
    <meta name="IS_Action_Pickup">
    <script language=javascript>
        var onHold = false;
        function ToggleHold() {
            if (onHold) {
                onHold = false;
                IS_Action_Pickup.click();
            } else {
                onHold = true;
                IS_Action_Hold.click();
            }
        }
    </script>
</head>
<body>
    <input type=button value="Hold" onclick="ToggleHold();">
</body>
```

IS_Action_Listen

Definition

This call control action allows an agent to listen to a call specified by the `callid` attribute. The listen action requires the `callid` of the call to listen in on. You would need to use the other queue functions in the scripter API to enumerate the call on another users' queue and retrieve one of the call ids and pass that into the listen action. Invoking the IS_Action_Listen element allows an agent to listen to another call; invoking the IS_Action_Listen element again turns off the listen feature. The current or active call of the current campaign in the queue is assumed to be the target of this action, if the CallId attribute is not set.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplte"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Listen element accepts the following attributes:

callid

> The `callid` to which this action applies (e.g. "89900001"). `callid` is required. A debug mode error is logged if the `callid` attribute is not specified.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Listen() {
  IS_Action_Listen.callid = document.getElementById("listenId").value;
  IS_Action_Listen.callback = function(error) {
    if (error) {
      console.error("IS_Action_Listen failed.");
    } else {
      console.log("IS_Action_Listen succeeded.");
    }
```

```
    }
}
```

Example 1

This example allows an agent to click the "Listen" button to listen to the call specified by the input Interaction ID.

```
<input type="text" id="listenId" placeholder="Interaction ID"> <button type="button"
onclick="listen()">Listen</button>  function listen() {

     IS_Action_Listen.callid = document.getElementById("listenId").value;

     IS_Action_Listen.click();

}
```

Example 2

This example connects to another agent's queue and automatically listens to any calls that are added to the queue.

```
<input type="text" id="user" placeholder="User"> <button type="button"
onclick="connectToQueueAndListen()">Listen to Queue</button>  function connectToQueueAndListen()
{

     var queue = scripter.createQueue();

     queue.callObjectAddedHandler = callObjectAdded;

     var user = document.getElementById("user").value;

     queue.connect(9, user);

}
function callObjectAdded(callObj) {

     IS_Action_Listen.callid = callObj.id;

     IS_Action_Listen.click();

}
```

IS_Action_Mute

Definition

This action mutes the current call, or the call specified by the `callid` attribute, so that the other party cannot hear what the agent says. Invoking the IS_Action_Mute element mutes a call; invoking the IS_Action_Mute element again turns off the mute feature. The current or active call of the current campaign in the queue is assumed to be the target of this action, if the CallId attribute is not set.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Mute element has the following attributes:

[callid]

> Optional. The `callid` to which this action applies (e.g. "89900001"). Dialer scripts use the current Dialer `callid` by default. A "debug mode" error is logged for non-Dialer scripts if the `callid` attribute is not specified.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Mute() {
  IS_Action_Mute.callback = function(error) {
    if (error) {
      console.error("IS_Action_Mute failed.");
    } else {
      console.log("IS_Action_Mute succeeded.");
    }
  }
}
```

Example 1

This is an example of a "Mute" button that allows the agent to mute a call so that the other party cannot hear the agent.

```
<body>
    <input type=button name="IS_Action_Mute" value="Mute">
</body>
```

Example 2

This is an example of a "Mute" button that invokes the "Mute" script function. The "mute" script function mutes a call so that the other party cannot hear the agent.;

```
<head>
    <meta name="IS_Action_Mute">
    <script language=javascript>
        function Mute() {
            IS_Action_Mute.click();
        }
    </script>
</head>
<body>
    <input type=button value="Mute" onclick="Mute();">
</body>
```

Example 3

This example shows how to mute the call specified by CallId.

```
<head>
    <meta name="IS_Action_Mute">
    <script language="javascript">
        function IS_MuteCall(p_page, p_CallId) {
            if (p_CallId != null) {
                IS_Action_Mute.callid = p_CallId;
            }
            IS_Action_Mute.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Mute Call" onclick="IS_MuteCall();">
</body>
```

IS_Action_Park

Definition

This call control action parks the current call, or a call identified by `callid`, on a user queue. The current call on the users' queue

is the default call that this action applies to. Syntactically, IS_Action_Park is similar to IS_Action_Transfer. When a call is parked, it is placed on hold and transferred to a local extension. IS_Action_Park assumes the current dialer call by default. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Park element has the following attributes:

[callid]

> Optional. The `callid` to which this action applies (e.g. "89900001"). Dialer scripts use the current Dialer `callid` by default. A "debug mode" error is logged for non-Dialer scripts if the `callid` attribute is not specified.

recipient

> Set this attribute to the user id or phone number of the call is being parked on.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Park() {
  IS_Action_Park.recipient = "BillS";
  IS_Action_Park.callback = function(error) {
    if (error) {
      console.error("IS_Action_Park failed.");
    } else {
      console.log("IS_Action_Park succeeded.");
    }
  }
}
```

Example 1

```
<head>
    <meta name="IS_Action_Park">
    <script language=javascript>
        function Park(userId) {
            IS_Action_Park.recipient = userId;
            IS_Action_Park.click();
        }
    </script>
</head> <body>
    <input type=button value="Park Call on Ext. 101" onclick='Park("101")'>
</body>
```

Example 2

Parks a call identified by its `callid`.

```
<head>
    <meta name="IS_Action_Park">
    <script language="javascript">
        function IS_ParkCall(p_userId, p_page, p_CallId) {
            if (p_CallId != null) {
                IS_Action_Park.callid = p_CallId;
            }
            IS_Action_Park.recipient = p_userId;
            IS_Action_Park.click();
```

```
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Park Call" onclick="IS_ParkCall(„101',null,null);">
</body>
```

IS_Action_Pickup

Definition

This action provides the ability to pick up the current call that is on the users' queue or answer a call specified by the `callid` attribute. Normally this action is called after a Hold action is performed. Calling this action takes the call off Hold. If the `callid` attribute is not set, the first active call in the queue is assumed to be the target of this action. If you call IS_Action_Pickup when there is no call, the routine terminates without generating an error.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Pickup element has the following attributes:

[callid]

Optional. The `callid` to which this action applies (e.g. "89900001"). CallId is optional. Dialer scripts use the current Dialer `callid` by default. A "debug mode" error is logged for non-Dialer scripts if the CallId attribute is not specified.

[callback]

The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Pickup() {
  IS_Action_Pickup.callback = function(error) {
    if (error) {
      console.error("IS_Action_Pickup failed.");
    } else {
      console.log("IS_Action_Pickup succeeded.");
    }
  }
}
```

Example 1

This is an example of a "Pickup" button that allows the agent to pick up a call.

```
<body>
    <input type=button name="IS_Action_Pickup" value="Pickup">
</body>
```

Example 2

This is an example of a "Pickup" button that invokes the "Pickup" script function. The "pickup" script function picks up a call.

```
<head>
    <meta name="IS_Action_Pickup">
    <script language=javascript>
```

```
        function Pickup() {
            IS_Action_Pickup.click();
        }
    </script>
</head>
<body>
    <input type=button value="Pickup" onclick='Pickup();">
</body>
```

Example 3

This example answers the call identified by `callid`.

```
<head>
    <meta name="IS_Action_Pickup">
    <script language="javascript">
        function IS_PickUpCall(p_page, p_CallId) {
            if (p_CallId != null) {
                IS_Action_Pickup.callid = p_CallId;
            }
            IS_Action_Pickup.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Pickup Call" onclick="IS_PickUpCall();">
</body>
```

IS_Action_PlaceCall

Definition

This action allows an agent to place another call from within the script that is loaded in Scripter. The call can be placed to another extension or an external phone number.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_PlaceCall element has the following attributes:

recipient

> The number of this recipient (e.g. "555-1212", or "101"). IS_Action_PlaceCall fails without a valid recipient.

[page]

> URL of page to open after the call is placed.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_PlaceCall() {
  IS_Action_PlaceCall.recipient = "555-1212";  IS_Action_PlaceCall.callback = function(error) {
    if (error) {
      console.error("IS_Action_PlaceCall failed.");
    } else {
```

```
              console.log("IS_Action_PlaceCall succeeded.");
          }
      }
  }
```

Example 1

This is an example of a "Place a Call" button that allows the agent to place a call (in this case, to extension 101).

```html
<body>
    <input type=button name="IS_Action_PlaceCall" value="Call Ext. 101" recipient="101">
</body>
```

Example 2

This is an example of a "Place a Call" button that invokes the "PlaceCall" script function. The "PlaceCall" script function places a call (in this case, to extension 101).

```html
<head>
    <meta name="IS_Action_PlaceCall">
    <script language=javascript>
        function PlaceCall(recipient) {
            IS_Action_PlaceCall.recipient = recipient.value;
            IS_Action_PlaceCall.click();
        }
    </script>
</head>
<body>
    <input type=button value="Call Ext. 101" onclick="PlaceCall('101');">
</body>
```

Example 3

```html
<head>
    <meta name="IS_Action_PlaceCall">
    <script language="javascript">
        function IS_PlaceCall(p_Number, p_Page) {
            IS_Action_PlaceCall.recipient = p_Number;
            IS_Action_PlaceCall.click();
            if (p_Page != null) location.href = p_Page;
        }
    </script>
</head>
<body>
    <input type=button value="Place Call" onclick="IS_PlaceCall('3178723000');">
</body>
```

IS_Action_PlaceChat

Definition

Initializes a chat between the current user and another user. To use this action, you must assign a callback function to IS_Action_PlaceChat.callback. This callback function is invoked as soon as the chat is initialized; it must contain all of the logic associated with the created chat interaction to ensure that execution does not proceed until the chat is fully initialized.

Attributes

recipient

> The userid of the other user that you would like to start a chat with.

Example

```
intitializeChat(userId)
{
  IS_Action_PlaceChat.recipient = userId;
  IS_Action_PlaceChat.click();
}
```

IS_Action_PlayWav

Definition

This call control action provides the ability to play a wave file to the call. Since the web page is hosted in Scripter, wave files can be loaded from the local machine without security issues, since the pages have access to the local file system on the workstation. Usage of this function requires a conference resource. Ensure conference resources have been allocated on the CIC Server. Without these resources, PlayWav will not function. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

> **NOTE:** For custom scripts in Scripter Connect, the wav file specified for the IS_Action_PlayWav action must be located in the Resource Path directory on the CIC server (I3\IC\Resources by default).

Attributes

The IS_Action_PlayWav element has the following attributes:

file

> Fully qualified path to a wave audio file. Scripts can optionally stream audio files from the ODS server by specifying http or https URI's.

[callid]

> Optional. The CallId to which this action applies (e.g. "89900001"). Dialer scripts use the current Dialer CallId by default. A "debug mode" error is logged for non-Dialer scripts if the CallId attribute is not specified.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_PlayWav() {
  IS_Action_PlayWav.file = "\\\\cic2\\recordings\\disclaimer.wav";
  IS_Action_PlayWav.callback = function(error) {
    if (error) {
```

```
              console.error("IS_Action_PlayWav failed.");
        } else {
              console.log("IS_Action_PlayWav succeeded.");
        }
      }
  }
```

Example 1

This example creates a button that plays a wave audio file. Note that extra backslashes must be added to the .wav filename, since Telephony Services escapes (removes) backslashes.

```html
<html>
<head>
    <meta name="IS_Action_PlayWav">
    <script language="javascript">
        function PlayDisclaimer() {
            IS_Action_PlayWav.file = "\\\\cic2\\recordings\\disclaimer.wav";
            IS_Action_PlayWav.click();
        }
    </script>
</head>
<body>
    <button onclick="PlayDisclaimer();">Play Disclaimer</button>
</body>
</html>
```

Example 2

```html
<head>
    <meta name="IS_Action_PlayWav">
    <script language="javascript">
        function IS_PlayWaveFileToCall(p_WaveFilePath, p_page) {
            IS_Action_PlayWav.file = p_WaveFilePath;
            IS_Action_PlayWav.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Place Wave" onclick="IS_PlayWaveFileToCall('C:\Temp\PartyOn.wav');">
</body>
```

IS_Action_Private

Definition

This call control action prevents other CIC users from listening to or recording the call specified by the `callid` attribute. The current or active call of the current campaign in the queue is assumed to be the target of this action, if the `callid` attribute is not set. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Private element has the following attributes:

[callid]

> Optional. The `callid` to which this action applies (e.g. "89900001"). Dialer scripts use the current Dialer `callid` by default. A "debug mode" error is logged for non-Dialer scripts if the `callid` attribute is not specified.

[callback]

The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Private() {
  IS_Action_Private.callback = function(error) {
    if (error) {
      console.error("IS_Action_Private failed.");
    } else {
      console.log("IS_Action_Private succeeded.");
    }
  }
}
```

Example 1

This is an example of a "Private" button that allows the agent to prevent other CIC users from listening or recording the call.

```
<body>
    <input type=button name="IS_Action_Private" value="Private">
</body>
```

Example 2

This is an example of a "Private " button that invokes the "Private" script function. The "private" script function prevents other CIC users from listening or recording the call.

```
<head>
    <meta name="IS_Action_Private">
    <script language=javascript>
        function Private() {
            IS_Action_Private.click();
        }
    </script>
</head>
<body>
    <input type=button value="Private" onclick="Private();">
</body>
```

Example 3

```
<head>
    <meta name="IS_Action_Private">
    <script language="javascript">
        function IS_PrivateCall(p_CallId, p_page) {
            if (p_CallId != null) {
                IS_Action_Private.callid = p_CallId;
            }
            IS_Action_Private.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Private Call" onclick="IS_PrivateCall();">
</body>
```

IS_Action_Record

Definition

This call control action provides the ability to record the current call or another call by passing in the `callid` of the call. When a record operation is invoked in this way, it functions like pressing the record button in a CIC client. If the user is associated with an email account, that user will receive a recording in their inbox. This action records the call specified by the `callid` attribute. The recording is saved as a .WAV file on the CIC Server. (See the online help file for the CIC client that you are using.) Selecting the IS_Action_Record element starts a recording session for a call; selecting the IS_Action_Record element again stops the recording session for a call. See also: IS_Action RecordPause.

All recordings associated with a call are saved in a single .WAV file. (There might be more than one recording associated with a call if the agent started and stopped the recording more than once during the call.) After the call is completed, the .WAV file can be attached to an email and sent to a CIC user, or stored in a database—see the Interaction Recorder online help for details concerning storage of recordings in a database. The current or active call of the current campaign in the queue is assumed to be the target of this action, if the `callid` attribute is not set.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Record element has the following attributes:

[callid]

> Optional. The `callid` to which this action applies (e.g. "89900001"). Dialer scripts use the current Dialer `callid` by default. A "debug mode" error is logged for non-Dialer scripts if the `callid` attribute is not specified.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Record() {
  IS_Action_Record.callback = function(error) {
    if (error) {
      console.error("IS_Action_Record failed.");
    } else {
      console.log("IS_Action_Record succeeded.");
    }
  }
}
```

Example 1

This is an example of a "Record" button that allows the agent to record a call.

```
<body>
    <input type=button name="IS_Action_Record" value="Record">
</body>
```

Example 2

This is an example of a "Record" button that invokes the "Record" script function. The "record" script function records a call.

```
<head>
    <meta name="IS_Action_Record">
    <script language=javascript>
        function Record() {
            IS_Action_Record.callid = IS_Attr_CallId.value;
            IS_Action_Record.click();
        }
    </script>
</head>
```

```
<body>
    <input type=button value="Record" onclick="Record();">
</body>
```

Example 3

```
<head>
    <meta name="IS_Action_Record">
    <script language="javascript">
        function IS_RecordCall(p_CallId, p_page) {
            if (p_CallId != null) {
                IS_Action_Record.callid = p_CallId;
            }
            IS_Action_Record.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Record Call"   onclick="IS_RecordCall();">
</body>
```

Example 4

This example toggles recording on and off

```
<head>
    <meta name="IS_Action_Record">
    <script language="javascript">
        function RecordCall(obj) {
            if (obj.value != "Recording...") {
                obj.value = "Recording...";
                IS_Action_Record.click();
            } else {
                obj.value = "Record Call";
                IS_Action_Record.click();
            }
        }
    </script>
</head>
<body>
    <input type=button id='rcrdCall' onclick="RecordCall(this);" value="Record Call">
</body>
```

IS_Action_RecordPause

Definition

This action allows an agent to pause recording of the call specified by the `callid` attribute. The current or active call of the current campaign in the queue is assumed to be the target of this action, if the `callid` attribute is not set. See also: IS_Action Record. This action works like a toggle. Selecting the IS_Action_RecordPause element pauses a recording session for the call; selecting the IS_Action_RecordPause again resumes the recording session for the call.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_RecordPause element has the following attributes:

[callid]

Optional. The `callid` to which this action applies (e.g. "89900001"). Dialer scripts use the current Dialer CallId by

Optional. The `callid` to which this action applies (e.g. "3998667"). Dialer scripts use the current Dialer CallId by default. A "debug mode" error is logged for non-Dialer scripts if the `callid` attribute is not specified.

[callback]

The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_RecordPause() {
  IS_Action_RecordPause.callback = function(error) {
    if (error) {
      console.error("IS_Action_RecordPause failed.");
    } else {
      console.log("IS_Action_RecordPause succeeded.");
    }
  }
}
```

Example 1

This is an example of a "Pause Recording" button that allows the agent to pause the recording of a call.

```
<body>
    <input type=button name="IS_Action_RecordPause" value="Pause Recording">
</body>
```

Example 2

This is an example of a "Pause Recording" button that invokes the "RecordPause" script function. The "RecordPause" function pauses the recording of a call.

```
<head>
    <meta name="IS_Action_RecordPause">
    <script language=javascript>
        function RecordPause() {
            IS_Action_RecordPause.click();
        }
    </script>
</head>
<body>
    <input type=button value="Pause Recording" onclick="RecordPause();">
</body>
```

Example 3

```
<head>
    <meta name="IS_Action_RecordPause">
    <script language="javascript">
        function IS_PauseRecord(p_CallId, p_page) {
            if (p_CallId != null) {
                IS_Action_RecordPause.callid = p_CallId;
            }
            IS_Action_RecordPause.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Pause Record Call" onclick="IS_PauseRecord();">
</body>
```

IS_Action_SelectPage

Definition

When multiple URLs are loaded in Scripter, HTML pages are displayed on separate tabs. This action brings focus to the calling page's tab in Scripter when an event occurs in that page.

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_SelectPage() {
  IS_Action_SelectPage.callback = function(error) {
    if (error) {
      console.error("IS_Action_SelectPage failed.");
    } else {
      console.log("IS_Action_SelectPage succeeded.");
    }
  }
}
```

Example

```html
<head>
    <meta name=IS_Action_SelectPage>
    <script language=javascript>
        window.onload = SelectPage;
        function SelectPage() {
            IS_Action_SelectPage.click();
        }
    </script>
</head>
```

IS_Action_SendToVoiceMail

Definition

This call control action allows an agent to send the call (specified by its `callid` attribute) to the agent's voice mail account. The current or active call of the current campaign in the queue is assumed to be the target of this action, if the `callid` attribute is not set. If the call is a predictive call, the call must still be dispositioned.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_SendToVoiceMail element has the following attributes:

[callid]

> Optional. The `callid` to which this action applies (e.g. "89900001"). Dialer scripts use the current Dialer `callid` by default. A "debug mode" error is logged for non-Dialer scripts if the CallId attribute is not specified.

[callback]

The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_SendToVoiceMail() {
  IS_Action_SendToVoiceMail.callback = function(error) {
    if (error) {
      console.error("IS_Action_SendToVoiceMail failed.");
    } else {
      console.log("IS_Action_SendToVoiceMail succeeded.");
    }
  }
}
```

Example 1

This is an example of a "Voice Mail" button that allows the agent to send a call to the agent's voice mail account.

```
<body>
    <input type=button name="IS_Action_SendToVoicemail" value="Voice Mail">
</body>
```

Example 2

This is an example of a "Voice Mail" button that invokes the "VoiceMail" script function. The "VoiceMail" script function sends a call to the agent's voice mail account.

```
<head>
    <meta name="IS_Action_SendToVoicemail">
    <script language=javascript>
        function Voicemail() {
            IS_Action_SendToVoicemail.click();
        }
    </script>
</head>
<body>
    <input type=button value="Voice Mail" onclick='Voicemail()'>
</body>
```

Example 3

```
<head>
    <meta name="IS_Action_SendToVoiceMail">
    <script language="javascript">
        function IS_SendCallToVoiceMail(p_page, p_CallId) {
            if (p_CallId != null)
                IS_Action_SendToVoiceMail.callid = p_CallId;
            IS_Action_SendToVoiceMail.click();
            if (p_page != null)
                location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="VoiceMaill" onclick="IS_SendToVoiceMail();">
</body>
```

IS_Action_SetForeground

Definition

This action brings the Scripter window to the top of all applications that are open on the desktop when an event happens in the page that is loaded in Scripter. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_SetForeground() {
  IS_Action_SetForeground.callback = function(error) {
    if (error) {
      console.error("IS_Action_SetForeground failed.");
    } else {
      console.log("IS_Action_SetForeground succeeded.");
    }
  }
}
```

Example

```
<head>
    <meta name="IS_Action_SetForeground">
    <script language=javascript>
        window.onload = SetForeground;
        function SetForeground() {
            IS_Action_SetForeground.click();
        }
    </script>
</head>
```

IS_Action_Trace

Definition

IS_Action_Trace writes an entry to the trace log, to aid in script debugging. This action adds custom trace messages from a custom script loaded in Scripter to Scripter's trace log file. The message is traced under the topic "Scripter Custom". Use this action when you need to troubleshoot an issue with a script or to compare the execution of the custom code in the script in relation to how Scripter executes the page that is loaded. The ability to trace is valuable when designing scripts for Scripter and should be taken into account for all scripts, whether they are inbound or outbound.

The first parameter is a string that contains the trace message. The second parameter is optional. It sets the tracing level and must be a value in the range (0-3). Trace messages generated by this mechanism have their own trace topic, "Scripter Custom".

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Trace element supports the following attributes:

message

Message is the trace message and is a string.

level

Level is the tracing level and must be one of the following:

0	Error
1	Warning
2	Status
3	Notes

If Level is invalid or missing, "Status" is used by default.

[callback]

The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Trace() {
  IS_Action_Trace.message = "Hello, World!.";  IS_Action_Trace.level = "3";
  IS_Action_Trace.callback = function(error) {
    if (error) {
      console.error("IS_Action_Trace failed.");
    } else {
      console.log("IS_Action_Trace succeeded.");
    }
  }
}
```

Example 1

```
<body>
    <input type=button name="IS_Action_Trace" Message="This is a test." Value="Trace">
</body>
<head>
    <meta name="IS_Action_Trace">
    <script language="javascript">
        function Trace(); {
            IS_Action_Trace.message = "This is a test.";
            IS_Action_Trace.level = "3";
            IS_Action_Trace.click();
        }
    </script>
</head>
<body>
    <input type=button onclick="Trace();" value="Trace">
</body>
```

Example 2

```
<head>
    <meta name="IS_Action_Trace">
    <script language="javascript">
      function IS_TraceNote(p_Message) {
          IS_Action_Trace.message = p_Message;
          IS_Action_Trace.level = 3; // 3= Notes level
          IS_Action_Trace.click();
      }
      function IS_TraceError(p_Message) {
          IS_Action_Trace.message = p_Message;
```

```
            IS_Action_Trace.level = 0; // 0= Error level
            IS_Action_Trace.click();
        }
    </script>
</head>
<body>
    <input type=button value="Trace Error Message" onclick="IS_TraceError('An Error Occurred');">
    <input type=button value="Trace Status Message" onclick="IS_TraceNote('A sale occurred');">
</body>
```

IS_Action_Transfer

Definition

This call control action transfers the call specified by the call ID attribute. It provides the ability to either do a blind transfer or a consult transfer on the current call that is on the users queue. The agent can either talk with the recipient before transferring the call (consult transfer) or transfer the call directly to the recipient without consultation (blind transfer). The current or active call of the current campaign in the queue is assumed to be the target of this action, if the CallId attribute is not set.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Transfer element has the following attributes:

callid

> The call ID to which this action applies (e.g. "89900001"). CallId is optional. Dialer scripts use the current Dialer CallId by default. A "debug mode" error is logged for non-Dialer scripts if the CallId attribute is not specified.

consult

> Flags that this transfer is the beginning of a two-stage consult transfer call (e.g. "True").

recipient

> If Consult call, the call ID of the recipient, otherwise the number of this recipient (e.g. "555-1212" or "101"). IS_Action_Transfer fails without a valid recipient. In the examples below, the agent who receives the transferred call must be running Interaction Scripter. If the call is transferred to an agent who is not running Interaction Scripter, the record must be dispositioned by the transferring agent first. See also: IS_Action_WriteData and Dialer disposition actions if you are writing an outbound call script.

audience

> This parameter is used with Interaction Connect scripts only. It toggles between participants of a consult transfer call. Use this parameter only after a consult transfer has been initiated. See Example 5 below.

neither	Places both parties of the conference on hold.
caller	Enables communication between the agent and the original party. The consulted party is placed on hold.
consultant	Enables communication between the agent and the consulted party. The original party is placed on hold.
both	Enables 3 way communication between the original party, agent, and consulted party.

> See also IS_Action_CompleteConsult.

callback

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, IS_Action_Transfer's callback property won't execute statements in its function code block until the transfer completes successfully.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Transfer() {
  IS_Action_Transfer.callid = 1234;
  IS_Action_Transfer.consult = False;
  IS_Action_Transfer.recipient = '377-522-2222';
  IS_Action_Transfer.callback = function(error) {
    if (error) {
      console.error("The Transfer action failed.");
    } else {
      console.log("The Transfer action was a success");
    }
  }
}
```

Statements inside the callback function block (highlighted above) execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Example 1

This is an example of a "Transfer to Ext. 101" button that transfers the current call to the user queue& at extension 101. This is an example of a blind transfer, where the call is transferred without consulting beforehand.

```
<body>
    <input type=buttonname="IS_Action_Transfer"value="Transfer to Ext. 101"consult="False"recipient="101">
</body>
```

Example 2

This is an example of a "Transfer to Ext. 101" button that invokes the "BlindTransfer" script function. The "BlindTransfer" script function transfers the current call to the specified number (in this example, extension 101). This is an example of a blind transfer, where the call is transferred without consulting beforehand.

```
<head>
    <meta name="IS_Action_Transfer">
    <script language="javascript">
        function IS_BlindTransfer(p_Number) {
            IS_Action_Transfer.recipient = p_Number;
            IS_Action_Transfer.consult = false;
            IS_Action_Transfer.click();
        }
    </script>
</head>
<body>
    <input type=button value="Transfer To 101" onclick="IS_BlindTransfer('101');">
</body>
```

Example 3

This example sets up a Consult transfer.

```
<head>
    <meta name="IS_Action_Transfer">
    <meta name="IS_Attr_CallId">
    <script language="javascript">
        function Transfer(number) {
            scripter.callObject.id = -1;
            scripter.callOjbect.dial(number, false);
            IS_Action_Transfer.consult = true;
            IS_Action_Transfer.recipient = scripter.callobject.id;
            Alert("Press OK when ready to continue");
            scripter.callObject.id = IS_Attr_CallId.value;
            scripter.callObject.pickup();
            IS_Action_Transfer.click();
        }
    </script>
</head>
<body>
```

```
    <input type=button value="Consult Transfer to 101" onclick="Transfer('101');">
</body>
```

Example 4

```
function ConsultTransfer(p_Number) {
    var p_mCallObj = scripter.createCallObject();
    var iRes1 = confirm("Would you like to call the 3rd party?");

    // user selected OK, so let's call the 3rd party
    if (iRes1) {
        p_mCallObj.dial(p_Number, false);
        // set up the consult transfer
        IS_Action_Transfer.consult = true;

        // set up the recipient call object
        IS_Action_Transfer.recipient = p_mCallObj.id;        var iRes2 = confirm("Press OK when you are ready
to transfer the call");

        //Transfer call to third party
        if (iRes2) {
            scripter.callObject.id = IS_ATTR_CallId.value;
            // pick up the call, it is probably on hold
            scripter.callObject.pickup();
            // now execute the Consult transfer that has been set up
            IS_Action_Transfer.click();
        } else {
            // they did not want to transfer, so lets disconnect the 3rd party
            // call and pick up the original call
            // disconnect 3rd party call
            p_mCallObj.disconnect();
            // pick up original call
            scripter.callObject.id = IS_ATTR_CallId.value;
            scripter.callObject.pickup();
        }
    }
}
```

Example 5 (Interaction Connect only)

This example shows how to use the audience parameter in a script for Interaction Connect to toggle between different legs of a consult transfer call.

```
IS_Action_Transfer.recipient = "1234";
IS_Action_Transfer.consult = true;
IS_Action_Transfer.click(); // A consult transfer is initiated. The agent is speaking with the consulted party
and the original party has been placed on hold.
IS_Action_Transfer.audience = "caller";
IS_Action_Transfer.click(); // The agent is now speaking with the original party, and the consulted party is
placed on hold.
IS_Action_Transfer.audience = "neither";
IS_Action_Transfer.click(); // Both parties of the conference are placed on hold.
IS_Action_Transfer.audience = "both";
IS_Action_Transfer.click(); // The agent is now speaking with both parties of the conference.
```

Predictive Actions

Interaction Scripter Predictive actions are only valid when the user is logged into Dialer. These actions are only applicable to Interaction Dialer campaigns, and therefore cannot be used in pages loaded into Scripter Client using the autoload command line option, or for inbound call handling. Predictive actions are also useful in preview mode, when information about a party is pushed to an agent before the agent initiates the call.

Action	Definition
IS_Action_BeginNonDialerCallScripting	Prevents the agent from being logged out of one campaign and into another while the agent is on a non-Dialer call.
IS_Action_CallComplete	Submits a call result to Dialer.
IS_Action_EndBreak	This action flags an agent's status so that the agent will receive campaign calls.
IS_Action_EndNonDialerCallScripting	Tells Scripter to resume automatic login of an Agent to a new campaign after transitioning was delayed by IS_Action_BeginNonDialerCallScripting.
IS_Action_EstablishPersistentConnection	This action calls the Agent only if a persistent connection has not been established. Afterwards it plays a .wav file to the Agent and then drops. The system keeps the audio connection open since the Agent has a persistent remote connection.
IS_Action_Logon	Logon to a campaign.
IS_Action_ManualOutboundCall	Initiates a preview call against an existing contact to either a supplied phone number, or to a different phone number from the contact list of the supplied record.
IS_Action_MarkCallForFinishing	Flags a call for redirection to a Finishing Agent.
IS_Action_PlacePreviewCall	Places a preview call (after a preview pop).
IS_Action_QueryContactList	Queries the specified contact list for records.
IS_Action_RequestBreak	Requests a break.
IS_Action_RequestContactData	Initiates a request to Dialer for additional contact columns associated with the current Dialer call along with PND table data for each contact column.
IS_Action_RequestLogoff	Requests logoff.
IS_Action_SkipPreviewCall	Skips a preview call (after a preview pop).
IS_Action_Stage	Move call to a new stage.
IS_Action_StartReceivingCalls	When Scripter Client is started with the /nostartreceiving parameter, a user can log into Interaction Scripter and set status to available, but Interaction Dialer will not place calls for the user until the IS_Action_StartReceivingCalls action is called. This parameter and its associated action are primarily used with Preview Campaigns.
IS_Action_TransferToAttendant	Transfers the active Dialer call to an outbound Attendant profile.
IS_Action_WriteData	Saves information associated with a predictive server to the predictive database.

IS_Action_BeginNonDialerCallScripting

Definition

NonDialerCallScripting is now used to delay closing a script when the agent has logged out of all the campaigns that the script is associated with. IS_Action_BeginNonDialerCallScripting prevents the agent from being logged out of one campaign and into another while the agent is on a non-Dialer call. Automatic campaign login will be delayed until the agent ends the non-dialer call. Use IS_Action_BeginNonDialerCallScripting to delay campaign transitions, and IS_Action_EndNonDialerCallScripting to terminate the delay. These are bracketing functions so that multiple calls to IS_Action_BeginNonDialerCallScripting() and IS_Action_EndNonDialerCallScripting() can be nested.

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_BeginNonDialerCallScripting() {
  IS_Action_BeginNonDialerCallScripting.callback = function(error) {
    if (error) {
      console.error("IS_Action_BeginNonDialerCallScripting failed.");
    } else {
      console.log("IS_Action_BeginNonDialerCallScripting succeeded.");
    }
  }
}
```

IS_Action_CallComplete

Definition

This action is used to disposition the current Dialer call. The details of the disposition will be determined by the attributes that are specified. Once this action has been initiated, no further changes can be made to the Dialer attributes. All element data is written to the server. If an update occurs to an element after this action, the update will be lost. This action does not disconnect the call after it has been dispositioned. You must use a separate action to disconnect the call.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_CallComplete element has the following attributes:

wrapupcode

> The wrap up code to be used for the disposition. This should be the full wrap up code e.g. Busy – Remote Busy.

[abandoned]

> The optional Abandoned attribute indicates even though the outbound call achieved call connect and routed to an Agent, the Dialer should consider it abandoned from a pacing, compliance and reporting perspective. The default value of this attribute is VARIANT_FALSE.

[agentid]

> When the scheduled party is called again, the call will be routed to the Agent identified by this call ID. If AgentId is not specified the call will be routed to any available agent.

callbacktime

Date and time that the targeted individual requests the agent to reschedule the campaign call. Typically, the date is entered in mm/dd/yy format and the time is entered in hh/mm AM/PM format (E.g. CallBackTime = 02/01/99 06:30 PM)

To specify a DATETIME format, set the CallBackTime attibute to a quoted text string, such as "8/12/2000 4:14 pm". This string must be normalized for your locale.

You can avoid the need to set CallBackTime to a DATETIME, by setting hour, minute, ampm, day, month, and year attributes instead. These optional attributes make it easier to set time formats, especially when the time format must be localized to a format other than US standard.

The HTML attributes below allow the script to set individual time values for the scheduled callback. These are Minute, Hour, Day, Month, Year, and AMPM. If CallBackTime is not used, each of the preceding attributes must be used.

Year	Year when callback will performed.
Month	Month when callback will performed.
Day	Day of month when callback will be performed.
Hour	Hour when callback will be performed.
Minute	Minute when callback will be performed.
AMPM	Optionally specifies AM or PM to indicate time of day. If this attribute is not specified, than a 24-hour format is assumed for Hour.

makefollowupcall

This Boolean indicates whether the user should be put into "Additional Follow Up" status, in support of a feature that allows an agent to dial additional calls while in that status. Setting this parameter True puts the agent in "Additional Follow Up" status, so that the agent can dial other contacts.

[callback]

The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_CallComplete(){
  IS_Action_CallComplete.wrapupcode = 'Scheduled';
  IS_Action_CallComplete.callbacktime = when;
  IS_Action_CallComplete.abandoned = false;
  IS_Action_CallComplete.callback = function(error) {
    if (error) {
      console.error("IS_Action_CallComplete failed.");
    } else {
      console.log("IS_Action_CallComplete succeeded.");
    }
  }
}
```

Example 1

This example dispositions a call with a wrap-up code and then disconnects the call.

```
<head>
    <meta name="IS_Action_CallComplete">
    <meta name="IS_Action_Disconnect">
    <script language="javascript">
        function EndCall(WrapUpCode) {
            IS_Action_CallComplete.wrapupcode = WrapUpCode;
            IS_Action_CallComplete.click();
```

```
        IS_Action_Disconnect.click();
    }
    </script>
</head>
<body>
    <input type="button" value="Call Successful" onclick=" EndCall ('Success')">
</body>
```

Example 2

In this example, a "Remove from Call List" button invokes the call completed action. The WrapUpCode attribute is populated with an appropriate value before the call is dispositioned.

```
<head>
    <meta name="IS_Action_CallComplete">
    <script language="javascript">
        function RemoveFromList(WrapUpCode) {
            IS_Action_CallComplete.wrapupcode = WrapUpCode;
            IS_Action_CallComplete.click();
        }
    </script>
</head>
<body>
    <input type="button" value="Remove from Call List" onclick="RemoveFromList('Deleted - Do Not Call')">
</body>
```

Example 3

This example demonstrates how to specify a callback time using the CallBackTime attribute:

```
<head>
    <meta name="IS_Action_CallComplete">
    <script language="javascript">
        function ScheduleCallback(when) {
            IS_Action_CallComplete.wrapupcode = 'Scheduled';
            IS_Action_CallComplete.callbacktime = when;
            IS_Action_CallComplete.abandoned = false;
            IS_Action_CallComplete.click();
        }
    </script>

    <body>
        <input type="button" value="Call Back" onclick="ScheduleCallback('03/15/2017 15:30');">
    </body>
```

IS_Action_EndBreak

Definition

This action provides the ability to send an end break to Dialer to make the agent available to take Dialer calls. This action is used in conjunction with the BeginBreak function. This action flags an agent's status so that the agent will receive campaign calls. Be careful. IS_Action_EndBreak does not change the agent's status from a DND (Do Not Disturb) state to an "available" state. Before calling IS_Action_EndBreak, you must set the agent's status to an available condition using IS_Action_ClientStatus.statusId. Otherwise, ACD calls will not be routed to the agent after he or she returns from the break. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_EndBreak() {
  IS_Action_EndBreak = function(error) {
    if (error) {
      console.error("IS_Action_EndBreak failed.");
    } else {
      console.log("IS_Action_EndBreak succeeded.");
    }
  }
}
```

Example 1

```html
<body>
    <input type=button name="IS_Action_EndBreak" value="Take More Calls">
</body>
```

Example 2

```html
<head>
    <meta name="IS_Action_EndBreak">
    <script language="javascript">
        function IS_EndBreak(p_availableStatus, p_page) {
            IS_Action_ClientStatus.statuskey = p_availableStatus;
            IS_Action_ClientStatus.click();
            IS_Action_EndBreak.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Go Available" onclick="IS_EndBreak('Available');">
</body>
```

IS_Action_EndNonDialerCallScripting

Definition

This action tells Scripter to resume automatic login of an Agent to a new campaign after transitioning was delayed by IS_Action_BeginNonDialerCallScripting. Use IS_Action_BeginNonDialerCallScriptingto delay transitions, and IS_Action_EndNonDialerCallScripting to terminate the delay. These are bracketing functions so that multiple calls to IS_Action_BeginNonDialerCallScripting() and IS_Action_EndNonDialerCallScripting() can be nested.

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_EndNonDialerCallScripting() {
  IS_Action_EndNonDialerCallScripting.callback = function(error) {
    if (error) {
      console.error("IS_Action_EndNonDialerCallScripting failed.");
    } else {
      console.log("IS_Action_EndNonDialerCallScripting succeeded.");
    }
  }
}
```

IS_Action_EstablishPersistentConnection

Definition

This action calls an Agent to establish a persistent connection. It has no effect unless the Agent is logged into Scripter using a Remote Station or Remote Number with the persistent connection setting set. If not, the action is ignored silently. It calls the Agent only if a persistent connection has not been established. Afterwards it plays a .wav file to the Agent and then drops. The system keeps the audio connection open since the Agent has a persistent remote connection. This establishes a persistent audio path before a campaign call is routed to an agent. There are two reasons for doing this:

- The called party does not experience a delay before the Agent gets connected to them (while the Agent's remote number is called the first time or whenever the persistent connection must be re-established).
- During this delay, the called party does not hear ringback. Outbound calls will not play ringback to the called party on the first call for a persistent remote Agent.

Script developers can hook this action into an initial page that is loaded only once, or they can invoke it from a button that Agents press. As an alternative, this action can be integrated into a break mechanism so that the connection is re-established whenever an Agent goes off break. Some customers routinely set up an initial script page that causes the Agent to be called to establish persistent connections. This built-in action simplifies this business practice.

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_EstablishPersistentConnection() {
  IS_Action_EstablishPersistentConnection.callback = function(error) {
    if (error) {
      console.error("IS_Action_EstablishPersistentConnection failed.");
```

```
  } else {
    console.log("IS_Action_EstablishPersistentConnection succeeded.");
  }
 }
}
```

Notes

The wave file that plays when a persistent connection is established is set on the Skills and ACD tab for a campaign, on the Persistent Connection expander.

This wave file should play a tone, or say something such as "connection established" to inform the agent that a persistent connection has been established. When agents logon to Interaction Scripter Client, the /initiate command line option can be used to initiate a persistent station connection. That option causes the system to invoke the IS_Action_EstablishPersistentConnection API action, which calls the agent to create a persistent audio path, before playing the wave audio to the agent.

Interaction Scripter Client login dialog

As a result, customers will no longer hear ringback on Dialer calls. Without this feature, the first call received by a remote agent causes the customer to hear ringback, because a connection to the remote station has to be established by having Telephony Services place a call to the agent. Afterwards, the remote station remains off hook and the receives calls without ringback. Calling the action eliminates ringback in all cases.

Example

```
<body>
    <input type=button name="IS_Action_EstablishPersistentConnection" value="Establish Persistent Connection">
</body>
```

IS_Action_Logon

Definition

This action can globally log the agent into Dialer or into the specified campaign. More specifically, the global logon performed by IS_Action_Logon will automatically log an agent into any campaigns that start after the global logon occurs. It will NOT log the agent into any currently running campaigns.

Attributes

The IS_Action_Logon action accepts these attributes:

[campaign]

> An optional attribute that can be used to specify a single campaign to log on to. If the Campaign attribute is used, the IS_Action_Logon will ONLY log the agent into the specified campaign.
> - If the agent has the Logon Campaign security right, the IS_Action_Logon action can NOT provide a global login.
> - IS_Action_Logon will not automatically log the agent into all currently running campaigns. In order to accomplish that, use the IS_Action_LogonAll action.
> - This action is normally used in conjunction with the RequestLogOff action to allow the user to log on and off of Dialer without using the menu options in Interaction Scripter.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Logon() {
  IS_Action_Logon.campaign = "ACME Campaign";  IS_Action_Logon.callback = function(error) {
    if (error) {
      console.error("IS_Action_Logon failed.");
    } else {
      console.log("IS_Action_Logon succeeded.");
    }
  }
}
```

Example 1

This action might be used by a "lobby" page script that allows selection of a campaign that then sends IS_Action_Logon to log the agent into a selected campaign.

```
IS_Action_Logon.campaign = "Newspaper Sales";
```

Example 2

```
<head>
    <meta name="IS_Action_Logon">
    <script language="javascript">
        function IS_LogOn() {
            IS_Action_Logon.click();
        }
    </script>

    <body>
        <input type=button value="Logon" onclick="IS_LogOn ();">
    </body>
```

IS_Action_LogonAll

Definition

This action globally logs the agent into Dialer as well as all currently running campaigns. In addition, this action will automatically log an agent into any campaigns that start after the global logon occurs. If the agent has the Logon Campaign security right, the IS_Action_LogonAll action can NOT provide a global login. However, it will log the agent into any currently running campaigns.

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_LogonAll() {
  IS_Action_Logon.callback = function(error) {
    if (error) {
      console.error("IS_Action_LogonAll failed.");
    } else {
      console.log("IS_Action_LogonAll succeeded.");
    }
  }
}
```

Example

```html
<head>
    <meta name="IS_Action_LogonAll">
    <script language="javascript">
        function IS_LogOnAll() {
            IS_Action_LogonAll.click();
        }
    </script>
</head>
<body>
    <input type=button value="Logon All" onclick="IS_LogOnAll ();">
</body>
```

IS_Action_ManualOutboundCall

Definition

This action will initiate a preview call against an existing contact to either a supplied phone number, or to a different phone number from the contact list of the supplied record.

Attributes

IS_Action_ManualOutboundCall accepts the following attributes:

i3identity

> The I3_Identity of the record in the contact table. This value is exposed by the IS_Attr_I3_IDENTITY attribute.

campaignid

> The unique Id of the campaign. This value is exposed by the IS_Attr_CampaignId attribute.

campaignname

> The name of the campaign. This value is exposed by the IS_Attr_CampaignName attribute.

contactcolumnId

> The Contact Column ID or CCID in the <ContactList>_CCD table. This value will be in the json string returned by the IS_Event_ContactDataLoaded event with the "CCID" property. Set this value to -1 if you wish to pass in a different phone number.

phonenumber

> The phone number you wish to dial if you set the ContactColumnId to -1.

overridemask

> This uses a bit mask to indicate which Scrub processes you wish to override during the manual outbound process. When you pass a number to the Manual Outbound Call Action, Dialer will then check number several ways to determine if it should be dialed. These checks include if the number is blocked by a Filter, or a Query Time Filter, if it is zone blocked, blocked because of skills, blocked for going over a daily limit, violating minimum spacing, blocked by a Do Not Call List, or blocked by campaign ownership. Each of these checks can be overridden by passing a series of flags that have been OR-ed together. The flag values are as follows:

```
var FLAG_Filter = 0x01;
var FLAG_QueryTimeFilter = 0x02;
var FLAG_ZoneBlocking = 0x04;
var FLAG_Skills = 0x08;
var FLAG_DailyLimit = 0x10;
var FLAG_MinimumSpacing = 0x20;
var FLAG_DNC = 0x40;
var FLAG_CampaignOwnership = 0x80;
```

> For example, IS_Action_ManualOutboundCall.OverrideMask = FLAG_Filter | FLAG_Skills] would prevent Dialer from scrubbing your number against Filters and Skills, while allowing the remaining checks to be active. If you wish to simply override everything, you can pass a value of 255 or 0xFF. The IS_Event_ManualOutboundCallStatus predictive event will also return a parameter called BlockedFlag that will contain one of these bit masks. You can then have a persistent variable that accumulates these flags and then pass it back to IS_Action_ManualOutboundCall after each attempt to override previous blocks. For example:

```
overrideParameter = overrideParameter | args.BlockedFlag ;
IS_Action_ManualOutboundCall.OverrideMask = overrideParameter ;
```

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_ManualOutboundCall() {
  IS_Action_ManualOutboundCall.callback = function(error) {
    if (error) {
      console.error("IS_Action_LogonAll failed.");
    } else {
      console.log("IS_Action_LogonAll succeeded.");
    }
  }
}
```

Example

See Example 1 in Sample Interaction Connect Scripts.

IS_Action_MarkCallForFinishing

Definition

This action flags a Dialer call for redirection to a Finishing Agent. It calls the IS_Action_WriteData.click() to update any new data gathered by the standard (opener) agent, and also calls the Predictive Dial COM API method named IEICPredictiveServer2::MarkCallForFinishing, which sends a request to Dialer. In turn, Dialer then sets the appropriate ACD category on the current call so that the interaction will be sent to a Finishing Agent of that campaign. The API call writes the data to the database.

> **Important**: do not use this action unless the *Maintain ACD Skills On Transfer* option is checked for the *System Configuration* in Interaction Administrator.
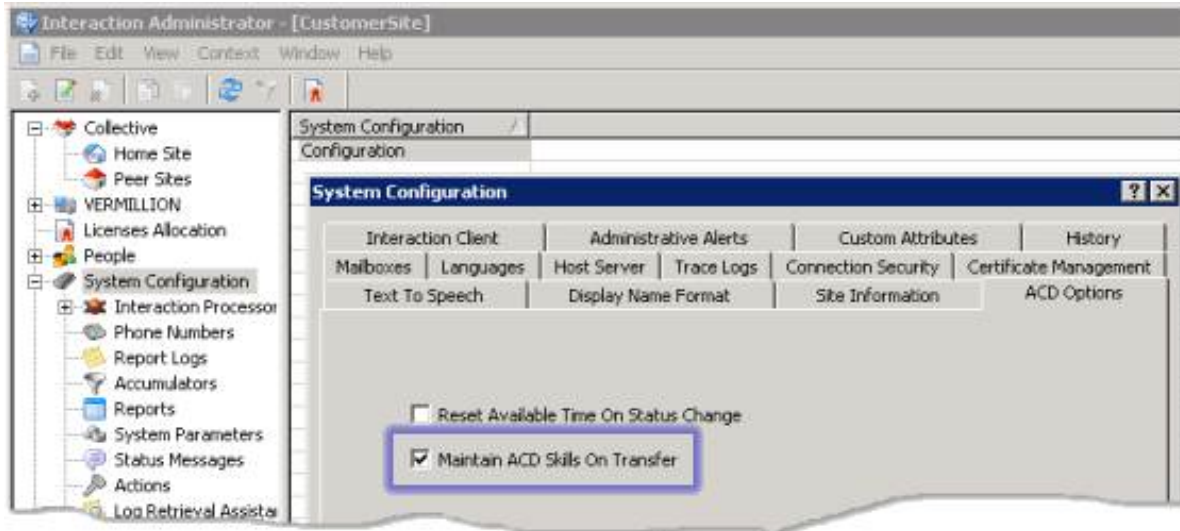
Maintain ACD Skills On Transfer is disabled by default. When checked, ACD categories are maintained when a call is transferred to a different workgroup or user. If this option is unchecked, it is possible for calls to be ACD routed to non-Finishing agents. This can happen because of the way that Dialer handles Finishing Agents:

- Finishing Agents log into the same ACD workgroup as regular agents. Dialer sets a special ACD category on them and later sets the same ACD category on each finishing call so that finishing calls are only be routed to finishing agents.
- When a regular agent transfers the finishing call to the ACD workgroup, the ACD categories are cleared as the call leaves the agent queue. Since the ACD categories have been cleared, the call is ACD routed to any agent in the workgroup, whether they are finishing or not. Checking the option mentioned above ensures that ACD categories are always maintained when a call is transferred, ensuring that only Finishing Agents will receive the call.

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_MarkCallForFinishing() {
  IS_Action_MarkCallForFinishing.callback = function(error) {
    if (error) {
      console.error("IS_Action_MarkCallForFinishing failed.");
    } else {
      console.log("IS_Action_MarkCallForFinishing succeeded.");
    }
  }
}
```

Example

```
<head>
```

```
        <meta name="IS_Action_MarkCallForFinishing">
        <meta name="IS_Action_Transfer">
        <script language="javascript">
            function IS_MarkCallForFinishing(p_Transfer) {
                IS_Action_MarkCallForFinishing.click();
                if (p_Transfer != null) {
                    IS_Action_Transfer.recipient = p_Transfer;
                    IS_Action_Transfer.consult = false;
                    IS_Action_Transfer.click();
                }
            }
        </script>
</head>
<body>
        <input type=button value="Mark Call For Finishing" onclick="IS_MarkCallForFinishing('4000');">
</body>
```

IS_Action_PlacePreviewCall

Definition

This action places a call that has been previewed by a campaign running in a Preview mode. In Preview mode, agents are presented with the next call record and given the choice to place the call, skip, reschedule, or delete the call record. IS_Action_PlacePreviewCall should be used when an agent presses a button to place the call. This action tells Dialer to place the call. Preview mode is sometimes used with third party applications and by customers who want to change the number that the preview call is going to place. Example 3 below indicates how to accomplish this. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_PlacePreviewCall() {
  IS_Action_PlacePreviewCall.callback = function(error) {
    if (error) {
      console.error("IS_Action_PlacePreviewCall failed.");
    } else {
      console.log("IS_Action_PlacePreviewCall succeeded.");
    }
  }
}
```

Example 1

```
<body>
        <input type=button name="IS_Action_PlacePreviewCall" value="Place Call">
</body>
```

Example 2

```
<head>
        <meta name="IS_Action_PlacePreviewCall">
        <script language=javascript>
            function PlacePreviewCall() {
```

```
            IS_Action_PlacePreviewCall.click();
        }
    </script>
</head>
<body>
    <input type=button value="Place Call" onclick='PlacePreviewCall()'>
</body>
```

Example 3

```
<head>
    <meta name="IS_Action_PlacePreviewCall">
    <meta name="IS_Attr_NumberToDial">
    <script language="javascript">
        function Place(p_Number) {
            IS_Attr_NumberToDial.value = p_Number;
            // wait 5 seconds, then place the call
            setTimeout('PlacePreviewCallExt()', 5000);
        }
        function PlacePreviewCallExt() {
            IS_Action_PlacePreviewCall.click();
        }
    </script>
</head>
<body>
    <input type=button value="Place Preview Call" onclick="Place(NumberToDial.value);">
    <input id="NumberToDial" name="NumberToDial" type="text" value="5554000" />
</body>
```

IS_Action_QueryContactList

Definition

Queries the specified contact list for records. Use with care. If used inappropriately this action could impede the performance of a script, or affect the performance of a database server.

Attributes

All of the following attributes are required:

statement

The sql query that will run against the specified contact list.

connectionid

The id of the connection associated with the contact list in the dialer_config.xml. The connectionId can be found in the dialer_config.xml by searching for a DIALEROBJECT with type="10". Below is an example of the default connection in dialer_config.xml.

```
<DIALEROBJECT id="{A0000000-0000-0000-0000-000000000000}" type="10" rev="2">
 <PROPERTIES>
  <udldataset>example.udl</udldataset>
  <dbms>0</dbms>
 </PROPERTIES>
</DIALEROBJECT>
```

displayname

The name of the contact list as defined in Interaction Administrator.

tablename

The database table name of the contact list.

callback

This function will be called once the query has returned. The associated records will be returned as a parameter of the callback.

Example

```
function queryContactList() {
  IS_Action_QueryContactList.tablename = "ContactListTable1";
  IS_Action_QueryContactList.displayname = "ContactList1";
  IS_Action_QueryContactList.connectionid = "{A0000000-0000-0000-0000-000000000000}";
  IS_Action_QueryContactList.statement = "select i3_identity, status, zone, phonenumber from
ContactListTable1";
  IS_Action_QueryContactList.callback = contactsReturnedCallback;
  IS_Action_QueryContactList.click();
}
function contactsReturnedCallback(contacts) {
  //The returned contacts could be used here for various purposes, such as populating a table with each
record.
}
```

IS_Action_RequestBreak

Definition

IS_Action_RequestBreak initiates a break request for the Agent. Dialer will check to see if other agents are available to handle any outstanding calls. If there are enough agents, the break request is granted. The last agent logged into a campaign is granted a break after pending call(s) for the agent are completed. When a break request is granted, the IS_Event_BreakGranted event is called so that the script may redirect the browser to a break page.

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_RequestBreak() {
  IS_Action_RequestBreak.callback = function(error) {
    if (error) {
      console.error("IS_Action_RequestBreak failed.");
    } else {
      console.log("IS_Action_RequestBreak succeeded.");
    }
  }
}
```

Example

This example creates a "Break" button that invokes the "IS_Action_RequestBreak" script function.

```
<head>
    <meta name="IS_Action_RequestBreak">
    <script language=javascript>
        function Break() {
            IS_Action_RequestBreak.click();
        }
    </script>
</head>
<body>
    <input type=button value="Break" onclick="Break();">
</body>
```

IS_Action_RequestContactData

Definition

This action will initiate a request to dialer for additional contact columns associated with the current dialer call along with PND table data for each contact column. When the data is loaded, the IS_Event_ContactDataLoaded predictive event is called, the data will be passed as a JSON string in the argument parameter.

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_RequestContactData() {
  IS_Action_RequestContactData.callback = function(error) {
    if (error) {
      console.error("IS_Action_RequestContactData failed.");
    } else {
      console.log("IS_Action_RequestContactData succeeded.");
    }
  }
}
```

Example

This example creates a Get Data button that invokes the IS_Action_RequestContactData script function.

```
<head>
    <meta name="IS_Action_RequestContactData">
    <script language=javascript>
        function RequestData() {
            IS_Action_RequestContactData.click();
        }
    </script>
</head>
<body>
    <input type=button value="Get Data" onclick="RequestData();">
</body>
```

IS_Action_RequestLogoff

Definition

This action requests an Agent logout. When the logoff request is granted, Interaction Scripter closes the outbound tab. It accepts an optional campaigns attribute, which can be used to request a logout for specific campaigns. If that attribute is not populated, then the action requests a logout for all campaigns. The format of the attribute is a list of campaign names separated by commas. If a campaign name contains a comma it can be escaped by encoding it for HTML e.g. `,`

Attributes

The IS_Action_RequestLogoff element has these attributes:

campaigns

> An optional list of campaign names separated by commas, to log out of.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_RequestLogoff() {
  IS_Action_RequestLogoff.campaigns = "Acme Collections";  IS_Action_RequestLogoff.callback =
function(error) {
    if (error) {
      console.error("IS_Action_RequestLogoff failed.");
    } else {
      console.log("IS_Action_RequestLogoff succeeded.");
    }
  }
}
```

Example

This example creates a "Logoff" button that invokes the "IS_Action_RequestLogoff" script function.

```html
<head>
    <meta name="IS_Action_RequestLogoff">
    <script language=javascript>
        function Logoff() {
            IS_Action_RequestLogoff.click();
        }
    </script>
</head>
<body>
    <input type=button value="Logoff" onclick="Logoff();">
</body>
```

IS_Action_SkipPreviewCall

Definition

This function is essentially equivalent to using IS_Action_CallComplete, but with a wrap up code of 'Skipped – Agent Skip'. This action skips a call that has been previewed. This action is used when the calling mode is set to *Preview*—whereby agents are presented with the next call record and given the choice to place the call, skip, reschedule, or delete the call record. IS_Action_SkipPreviewCall should be used when an agent presses a button to skip the call. It tells Dialer that another preview call is needed.

This action does not make the record uncallable under any circumstances. Once IS_Action_SkipPreviewCall has been initiated, no further changes can be made to the Dialer attributes. All element data is written to the server. If an update occurs to an element after this action, the update will be lost. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_SkipPreviewCall element has the following attributes:

wrapupcode

> The wrap up code to be used for the disposition. This should be the full wrap up code e.g. Skipped - Agent Skip.

[abandoned]

> The optional Abandoned attribute indicates even though the outbound call achieved call connect and routed to an Agent, the Dialer should consider it abandoned from a pacing, compliance and reporting perspective. The default value of this attribute is false.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_SkipPreviewCall() {
  IS_Action_SkipPreviewCall.wrapupcode = "Skipped - Agent Skip";  IS_Action_SkipPreviewCall.callback =
function(error) {
    if (error) {
      console.error("IS_Action_SkipPreviewCall failed.");
    } else {
      console.log("IS_Action_SkipPreviewCall succeeded.");
    }
  }
}
```

Example 1

This is an example of a "Skip Call" button that skips the Preview call that has been presented to the agent.

```
<body>
    <input type=button name="IS_Action_SkipPreviewCall" value="Skip Call">
</body>
```

Example 2

This is an example of a "Skip Call" button that invokes the "SkipCall" script function. The WrapUpCode attribute is manually populated with the appropriate value before the call is dispositioned.

```
<head>
    <meta name="IS_Action_SkipPreviewCall">
    <script language=javascript>
        function SkipCall() {
            IS_Action_SkipPreviewCall.wrapupcode = "Skipped - Agent Skip";
            IS_Action_SkipPreviewCall.click();
        }
    </script>
</head>
<body>
    <input type=button value="Skip Call" onclick='SkipCall()'>
</body>
```

Example 3

```html
<head>
    <meta name="IS_Action_SkipPreviewCall">
    <script language="javascript">
        function IS_SkipPreviewCall(p_page) {
            IS_Action_SkipPreviewCall.wrapupcode = "Skipped - Agent Skip";
            IS_Action_SkipPreviewCall.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value="Skip Preview Call" onclick="IS_SkipPreviewCall(null);">
</body>
```

IS_Action_Stage

Definition

This action is used to set stages within a predictive campaign script. It moves a call to a new stage. Staging is normally only used in predictive campaigns, it does not apply to power or preview campaigns. The staging values correlate with the values set in the staging container in Interaction Administrator that is assigned to the active campaign. A campaign call can be segmented into various stages that an agent may traverse during a campaign call. While a campaign is active, Interaction Dialer monitors agent performance per stage and maintains values that allow Outbound Server to predict the probability of the agent finishing the call in that stage, as well as how long the agent is expected to be in that stage. For example, the following table depicts how a sample telemarketing campaign might be staged:

Id	Name	Probability	Adjusted Call Length
Stage 1	Introduction	67%	37 seconds
Stage 2	Preliminary Sales Pitch	75%	1 minute, 42 seconds
Stage 3	Detailed Product Description	80%	5 minutes, 48 seconds
Stage 4	Billing information	100%	2 minutes, 30 seconds

The Probability Value is the likelihood of the call ending in that stage for a particular agent. Each agent has a table of values (as in the example above) that corresponds to that agent's personal statistical summary while the agent is logged into a campaign. Therefore, if one agent takes longer in a particular stage than another agent, the algorithm adjusts accordingly.

> **Tip**—the topic titled 'Stage Sets view' in Interaction Dialer Manager Help explains how to define stages.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

The IS_Action_Stage element has the following attribute:

stage

> The call stage number to set. The call stage number can be any number in the range of 0-10,000. We recommend that you make stage numbers consecutive. (E.g. 1, 2, 3, 4, 5; not 1, 2, 5, 20, 45). This number should match the value of a stage from the Stages associated with the campaign. It cannot match the stage name, only the numeric value. If no matching stage value is found, Interaction Dialer will choose a stage name (e.g.: "Auto-Added Stage #), where # is the specified stage value.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_Stage() {
  IS_Action_Stage.stage = "3";  IS_Action_Stage.callback = function(error) {
    if (error) {
      console.error("IS_Action_Stage failed.");
    } else {
      console.log("IS_Action_Stage succeeded.");
    }
  }
}
```

Example 1

This is an example of a button that moves a call to the next stage (which, in this case, is Stage 1).

```
<body>
    <input type=button name="IS_Action_Stage" value="Stage 1" Stage="1">
</body>
```

Example 2

This is an example of a button that invokes the "Stage" script function. The "Stage" script function moves a call to the next stage (which, in this case, is Stage 3).

```
<head>
    <meta name="IS_Action_Stage">
    <script language=javascript>
        function Stage(stageNumber) {
            IS_Action_Stage.stage = stageNumber;
            IS_Action_Stage.click();
        }
    </script>
</head>
<body>
    <input type=button value="Stage 3" onclick='Stage("3")'>
</body>
```

Example 3

```
<head>
    <meta name="IS_Action_Stage">
    <script language="javascript">
        function IS_SetCurrentStage(p_page, p_stage) {
            IS_Action_Stage.stage = p_stage;
            IS_Action_Stage.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>
<body>
    <input type=button value=" Next Page" onclick="IS_SetCurrentStage ('ClosingSale.htm', 4);">
</body>
```

IS_Action_StartReceivingCalls

Definition

When Scripter Client is started with the /nostartreceiving optional command line parameter, an agent can log into Interaction Scripter and set status to available, but Interaction Dialer will not place calls for that agent until the IS_Action_StartReceivingCalls action is called. This feature is used primarily with Preview campaigns.

- Scripts should not call this action unless the /nostartreceving scripter command line option is used.
- IS_Action_StartReceivingCalls should only be called once in the script when the agent or script is first ready to receive calls. Calling it more than once can cause problems.
- If you want the IS_Action_StartReceivingCalls Predictive action to function on a per campaign basis rather than on a per agent basis, you will need to use the Dialer StartReceivingCalls Per Campaign server parameter with the value set to 1. Keep in mind that this action only works for agents who have the Logon Campaign right.
- For additional information, see the Server Parameters topic in the Dialer Manager Help system.

Attributes

[campaigns]

> An optional, comma separated list of campaign names to notify Dialer the agent is ready to start receiving calls from. To use this attribute, the "Dialer StartReceivingCalls Per Campaign" server parameter must to be enabled.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.
>
> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_StartReceivingCalls() {
  IS_Action_StartReceivingCalls.callback = function(error) {
    if (error) {
      console.error("IS_Action_StartReceivingCalls failed.");
    } else {
      console.log("IS_Action_StartReceivingCalls succeeded.");
    }
  }
}
```

Example

```
<head>
    <meta name="IS_Action_StartReceivingCalls">
    <script language=javascript>
        function
        StartReceivingCalls() {
            IS_Action_StartReceivingCalls.click();
        }
    </script>
</head>
<body>
    <input type=button value="StartReceivingCalls" onclick="StartReceivingCalls();">
</body>
```

Example 2

```
IS_Action_StartReceivingCalls.campaigns = "Campaign1,Campaign2,Campaign3";
IS_Action_StartReceivingCalls.click();
```

IS_Action_TransferToAttendant

Definition

Transfers the active Dialer call to an outbound Attendant profile.

Attributes

attendantprofile

> The name of the outbound Attendant profile to which the call will be transferred.

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_TransferToAttendant() {
  IS_Action_TransferToAttendant.attendantprofile = "Outbound
IVR";  IS_Action_TransferToAttendant.callback = function(error) {
    if (error) {
      console.error("IS_Action_TransferToAttendant failed.");
    } else {
      console.log("IS_Action_TransferToAttendant succeeded.");
    }
  }
}
```

Example

```
IS_Action_TransferToAttendant.attendantprofile = "Outbound IVR";
IS_Action_TransferToAttendant.click();
```

IS_Action_WriteData

Definition

This action saves information associated with a predictive call to the Dialer cache. Use this action before transferring a call, if the agent is transferring to a supervisor or Finishing Agent and MarkCallForFinishing is not used. This allows data to be updated in cache before the transfer action is performed so that the new agent will receive the updated attributes. You do not need to call IS_Action_WriteData at the end of each call, unless the call is to be transferred. Interaction Scripter automatically saves predictive data when navigating to a new page. The IS_Action_WriteData function should not be called before navigating between pages. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Attributes

[callback]

> The callback property ensures that this action executes asynchronously in Interaction Connect. Starting with 2018 R3, all Interaction Scripter actions (IS_Actions) provide a callback property for use in Connect scripts only. In the example below, statements inside the highlighted callback function block execute only after the action completes. The callback will return an error if the action fails. See Writing custom scripts for Interaction Connect or Scripter .NET.

> Here's how to use the .callback property in a script for Interaction Connect:

```
function IS_Action_WriteData() {
  IS_Action_WriteData.callback = function(error) {
```

```
    if (error) {
      console.error("IS_Action_WriteData failed.");
    } else {
      console.log("IS_Action_WriteData succeeded.");
    }
  }
}
```

Example 1

```html
<head>
    <meta name="IS_Action_WriteData">
</head>
<body>
    <input name="IS_Action_Transfer" type=button value="Transfer to Supervisor" consult="false"
recipient="101" onclick="IS_Action_WriteData.click();">
</body>
```

Example 2

```html
<head>
    <meta name="IS_Action_Transfer">
    <meta name="IS_Action_WriteData">
    <script language=javascript>
        function SupervisorTransfer() {
            IS_Action_WriteData.click();
            IS_Action_Transfer.recipient = "101";
            IS_Action_Transfer.consult = "false";
            IS_Action_Transfer.click();
        }
    </script>
</head>
<body>
    <input type=button value="Transfer to Supervisor" onclick='SupervisorTransfer()'>
</body>
```

Example 3

```html
<head>
    <meta name="IS_Action_WriteData">
    <script language="javascript">
        function IS_WriteData(p_page) {
            IS_Action_WriteData.click();
            if (p_page != null) location.href = p_page;
        }
    </script>
</head>

<body>
    <input type=button value="Update Data" onclick="IS_WriteData();">
</body>
```

Interaction Scripter Events

Events are notification messages from the CIC server that trigger script functions. For example, an event can provide notification that a queue on the server has changed. When a call is placed on a queue, this event changes the queue, generating an event message.

Standard Events

Standard events are generalized and can be used in any script, including scripts for blended environments. These events are not directly associated with being logged into Dialer, though they can be used when logged into Dialer too. Standard events are generated when queues change.

Predictive Events

Predictive events are notification events associated with campaign activities for Predictive, Power or Preview campaigns. Predictive events are raised by Scripter when an agent is logged into Dialer. All predictive events are functions declared in a script that are called when an event occurs.

Standard Events

Standard events are generated when queues change. Interaction Scripter standard events are generalized and can be used in a script when implementing a blended environment. These events are not directly associated with being logged into Dialer, though they can be used when logged into Dialer too.

Event	Definition
IS_Event_QueueObjectAdded	This event occurs when a queue object has been added to the user's queue. If your application's interface displays a call queue, this event provides notification that a call should be added to the list.
IS_Event_QueueObjectChanged	This event provides notification that the state of a queue object or call object has changed in the user's queue.
IS_Event_QueueObjectRemoved	This event occurs when a queue object is removed from the user's queue. If your application's interface displays a call queue, this event provides notification that a call can be removed from the list.

IS_Event_QueueObjectAdded

Definition

This event is raised by Scripter when a queue object is added to the user's queue. Scripter will listen for calls and chats being added to the users' queue. If your application's interface displays a call queue, IS_Event_QueueObjectAdded provides notification that a call should be added to the list.

Attributes

The QueueName and ObjectId parameters are optional. If you use them, you must specify both parameters.

QueueName

> The name of the call queue.

ObjectId

> The object id of the new object in the queue.

Syntax

```
function IS_Event_QueueObjectAdded(queueName, ObjectId)
```

```
function IS_Event_QueueObjectAdded()
```

Example

```html
<head>
    <script language=javascript>
        function IS_Event_QueueObjectAdded(queueName, ObjectId) {

            // add an object to the queue display
            // insert other code here as needed...

        }
    </script>
</head>
```

IS_Event_QueueObjectChanged

Definition

This event provides notification that the state of a queue object or call object has changed in the user's queue. This event is raised by Scripter when the object that was added on the user's queue changes state. For example the call may go from a "Connected" state to a "Hold" state after issuing a IS_Action_Hold on the call.

Attributes

The QueueName and ObjectId parameters are optional. If you use them, you must specify both parameters.

QueueName

> The name of the call queue.

ObjectId

> The object id of the object being changed.

Syntax

```
function IS_Event_QueueObjectChanged(QueueName, ObjectId)

function IS_Event_QueueObjectChanged()
```

Example

```html
<head>
    <script language=javascript>
        function IS_Event_QueueObjectChanged(QueueName, ObjectId) {

            // update the queue object displayed on the screen
            // insert other code here as needed...

        }
    </script>
</head>
```

IS_Event_QueueObjectRemoved

Definition

This event occurs when a queue object is removed from the user's queue. If your application's interface displays a call queue, IS_Event_QueueObjectRemoved provides notification that a call can be removed from the list.

This event is raised by Scripter when the object is either destroyed, (usually 2 minutes after the call is disconnected), or when the call is transferred off of the user's queue, such as when transferred to a finishing agent.

Attributes

The QueueName and ObjectId parameters are optional. If you use them, you must specify both parameters.

QueueName

    The name of the call queue.

ObjectId

    The object id of the call being removed.

Syntax

```
function IS_Event_QueueObjectRemoved(QueueName, ObjectId)

function IS_Event_QueueObjectRemoved()
```

Example 1

```
<head>
    <script language=javascript>
        function IS_Event_QueueObjectRemoved(QueueName, ObjectId) {

            // remove this object from the queue display
            // insert other code here as needed...

        }
    </script>
</head>
```

Example 2

This example can be used as an include file in a web page to watch for the above events and update the interface as needed.

```
<head>
    <script language="javascript">
        // global call object
        var mg_callObj = scripter.createCallObject();
        function IS_Event_QueueObjectAdded(p_QueueName, p_ObjectId) {
            mg_callObj.Id = p_ObjectId.
        }
        function IS_Event_QueueObjectChanged(p_QueueName, p_ObjectId) {

            // current object state
            alert(mg_callObj.stateString);
        }
        function IS_Event_QueueObjectRemoved(p_QueueName, p_ObjectId) {
            alert('Object Removed');
        }
    </script>
</head>
```

Predictive Events

Predictive Events are notification messages from the server, such as a new call on a queue. All predictive events are functions that you declare and are called when an event occurs. Predictive events are events associated with campaign activities. They are usually associated with Predictive, Power or Preview campaigns and only get raised by Scripter when an agent is logged into Dialer.

Event	Definition
IS_Event_BreakGranted	This event tells a script that an IS_Action_RequestBreak request has been granted, so that the script can redirect the browser to a break page.
IS_Event_ContactDataLoaded	This event is specifically designed for use with the IS_Action_RequestContactData action.
IS_Event_DataPop	This event is fired in response to new incoming data from a predictive or preview call. This event is only called if there is no IS_Event_NewPredictiveCall event handler defined (for preview if no IS_Event_PreviewDataPop or if no IS_Event_NewPreviewCall).
IS_Event_ManualOutboundCallStatus	This event is specifically designed for use with the IS_Action_ManualOutboundCall action.
IS_Event_NewPredictiveCall	This event is fired when a new predictive call is placed in a queue. Use the IS_Attr_CampaignName attribute to identify the name of the campaign.
IS_Event_NewPreviewCall	This event is fired when a new preview call is placed in a queue.
IS_Event_PreviewCallSkipped	This event is emitted when a preview call is successfully skipped.
IS_Event_PredictiveCallReleased	This event is generated when a call is disconnected, transferred, or stolen from the call queue.
IS_Event_PreviewDataPop	This event is useful in preview dialing mode. IS_Event_PreviewDataPop provides notification that the client can display a customer record, before the call is placed. This allows the agent to review the client record before pushing a button to initiate the call.
PreviewTimeout Events	There are three PreviewTimeout events. These are distinctive Predictive Events that are specifically designed for use with Preview campaigns that use a preview countdown timer.

IS_Event_BreakGranted

Definition

This event tells a script that a IS_Action_RequestBreak request has been granted, so that the script can redirect the browser to a break page. Dialer grants break requests if other agents available to handle any outstanding calls. The last agent logged into a campaign is granted a break after all pending call(s) for the agent are completed.

Attributes

None.

Example

```html
<head>
    <script language=javascript>
        function IS_Event_BreakGranted() {
            // navigate to the on break page
            location.href = "AgentBreak.html";
        }
    </script>
</head>
```

IS_Event_ContactDataLoaded

Definition

The IS_Event_ContactDataLoaded event is a Predictive Event that is specifically designed for use with IS_Action_RequestContactData. When Dialer has acquired all the contact Data for the current record it will send the IS_Event_ContactDataLoaded event to a custom script along with a JSON string containing all the data as an argument.

Attributes

JsonString

A JSON formatted string containing all the data from the PND table, along with the Contact Column Name, Contact Column ID, Phone Number and the phone number Type. If an exception is thrown while attempting to load the data, there will be an error property instead of data returned. This error can be trapped and logged or displayed on the script.

Example

```html
<head>
    <script language=javascript>
        var TraceLevel = {
            Error: 0,
            Warning: 1,
            Status: 2,
            Note: 3
        };
        function IS_Event_ContactDataLoaded(json) {
            scripter.trace(json, TraceLevel.Note)
            var obj = eval("(" + json + ")");
            if (obj.error) {
                scripter.trace('error = ' + obj.error, TraceLevel.Error);
            }
            var name = obj.NAME;
            var ccid = obj.CCID;

            // insert other code here as needed...
        }
    </script>
</head>
```

Sample JSON

```json
[{
"BLOCKINGRESULT":"0",
"CCID": 575,
"I3_ATTEMPTS": 0,
"I3_ATTEMPTSABANDONED": 0,
"I3_ATTEMPTSBUSY": 0,
"I3_ATTEMPTSFAX": 0,
"I3_ATTEMPTSMACHINE": 0,
"I3_ATTEMPTSNOANSWER": 0,
"I3_ATTEMPTSREMOTEHANGUP": 0,
"I3_ATTEMPTSRESCHEDULED": 0,
"I3_ATTEMPTSSITCALLABLE": 0,
"I3_ATTEMPTSSYTEMHANGUP": 0,
"I3_DNCCOMEXPIRATION": "",
"I3_STATUS": "",
"I3_SUBSTATUS": "",
"I3_ZONE": "",
"NAME": "PHONE",
"PHONENUMBER": "3175557188",
"PHONENUMBERTYPE": "",
"Sex": "",
```

```
"State": "",
"TimeZone": ""
}]
```

Dialer will pre scrub numbers for zone blocking and DNC blocking. If the numbers meet either of these conditions, the BLOCKINGRESULT parameter will have a value that corresponds to the blocked flags. (See IS_Event_ManualOutboundCallStatus BlockedFlag attribute) You can submit that blocked flag back to IS_Action_ManualOutboundCall as the override parameter and the call will be placed. If you try to place a call that has a BLOCKINGRESULT value without submitting the override parameter, the call will fail as invalid.

IS_Event_DataPop

Definition

This event is fired in response to new incoming data from a predictive or preview call. Its prototype is IS_Event_DataPop(Names, Values). This event is only called if there is no IS_Event_NewPredictiveCall event handler defined (for preview if no IS_Event_PreviewDataPop or if no IS_Event_NewPreviewCall). This event is raised when the data that is associated with the predictive or power dialed call is presented to the agent. This event is passed a JavaScript array of names and values representing the column names and column values in the call list the campaign is dialing from.

Attributes

Names

A JavaScript array of the Dialer attribute names.

Values

A JavaScript array of the Dialer attribute values.

Example 1

```html
<head>
    <meta name="IS_Action_SelectPage">
    <meta name="IS_Action_SetFocus">
    <script language=javascript>
        function IS_Event_DataPop(Names, Values) {

            // move this tab page to the top and set
            // focus to this application window

            IS_Action_SelectPage.click();
            IS_Action_SetFocus.click();
        }
    </script>
</head>
```

Example 2

This sample script iterates through the JavaScript array and writes out the column name and its corresponding value.

```html
<head>
    <script language="javascript">
        // global call object
        function IS_Event_DataPop(p_Names, p_Values) {
            var x;
            for (x in p_Names) {
                document.write('Column Name: >'+p_Names[x] + ' -- - Column Value: '+p_Values[x]);
            }
        }
    </script>
</head>
```

IS_Event_ManualOutboundCallStatus

Definition

The IS_Event_ManualOutboundCallStatus event is a predictive event that is specifically designed for use with the IS_Action_ManualOutboundCall predictive action. When Dialer has finished processing the Manual Outbound Call, it will send the IS_Event_ManualOutboundCallStatus event to a custom script.

Attributes

The IS_Event_ManualOutboundCallStatus event has four attributes that are passed in as an argument object.

Identity

> This is a string that holds the unique identifier of the record as defined in the contact list table, I3_IDENTITY.

StatusName

> A string that indicates the status of manual outbound call processing: `CallPlaced, CallComplete, PolicyCompleted, CallBlocked, PreviouslyDialed, ContactBlocked, ContactNotFound, ContactUncallable, InvalidPhoneNumber, InvalidCampaign, InvalidAgent, AgentNotIdle, InternalError, or PlaceCallFailed.`

Status

> Status returns a number corresponding to the StatusName string.

UncallableStatus

> This is the status value stored in the contact table if the status is ContactUncallable.

CallBlockedDescriptionString

> If the call is blocked, then this string will indicate the reason the call was blocked as a localized string that you can pass to your script.

BlockedFlag

> This is the value of the flag that was blocked.

Dec	Hex	Meaning
1	0x01	Blocked by Filter
2	0x02	Blocked by Query Time Filter
4	0x04	Blocked by Time Zone
8	0x08	Blocked by skills
16	0x10	Blocked by the Daily Limit value
32	0x20	Blocked by minimum spacing between dials
64	0x40	Blocked by Do Not Call rules
128	0x80	Blocked by campaign ownership

> This value can be passed back to IS_Action_ManualOutboundCall as the overridemask in order to make the call again, but to ignore the rule that was just blocked. If you OR multiple Flags together, you can override multiple checks against the number.

Example

```
var OverrideCode = {
```

```javascript
    "None": 0x00,
    "Filter": 0x01,
    "QueryTimeFilter": 0x02,
    "Zone": 0x04,
    "Skills": 0x08,
    "DailyLimit": 0x10,
    "MinimumSpacing": 0x20,
    "PNDStatus": 0x40,
    "DNC": 0x80,
    "CampaignOwndership": 0x100
};
var TraceLevel = {
    Error: 0,
    Warning: 1,
    Status: 2,
    Note: 3
};
var overrideAccumulator = 0x0;
var lastWrapUpCode = "";
var dialedNumber;
var calledNumberArray = [];
function EndCall(WrapUpCode) {
    if (WrapUpCode == 'Failure') {
        var answer = confirm("Would you like to call another number?")
        if (answer) {
            calledNumberArray.push();
            lastWrapUpCode = WrapUpCode;
            IS_Action_RequestContactData.click();
            scripter.trace("ContactDataRequested", TraceLevel.Note);
            // Once RequestContactData is called, a response will
            // trigger IS_Event_ContactDataLoaded
            return;
        }
    }
    CalledNumberArray = [];
    CompleteCall(WrapUpCode);
}
function CompleteCall(WrapUpCode) {
    IS_Action_CallComplete.WrapUpCode = WrapUpCode;
    IS_Action_CallComplete.click();
    IS_Action_Disconnect.click();
}
function IS_Event_ContactDataLoaded(args) {
    scripter.trace(args,
        TraceLevel.Note) // Raw json string will be traced to the log.
    var contactDataObject;
    try {
        JSON.parse(args, contactDataObject); // This does not work in older IE browsers;
    } catch (err) {
        contactDataObject = eval("(" + args + ")");
        // JSON is not supported in older iE browsers.
    }
    // If there is an error on the server, the result object will have an error property.
    if (contactDataObject.error) {
        scripter.trace('error= ' + contactDataObject.error, TraceLevel.StatusName);
    }
    var NextContactToCall;
    for (var i = 0; i < contactDataObject.length; i++) {
        if ($.inArray(contactDataObject[i].PHONENUMBER, calledNumberArray))
            continue;
        if (contactDataObject[i].OVERRIDECODE & OverrideCode.Zone == OverrideCode.Zone) {
            var answer = confirm("This number is blocked by Time Zone Rules, would you still like to dial
it?")
            if (!answer)
                continue;
            else
                overrideAccumulator = overrideAccumulator | OverrideCode.Zone
        }
        if (contactDataObject[i].OVERRIDECODE & OverrideCode.DNC == OverrideCode.DNC) {
            var answer = ok("This number is on the Do Not Call List, would you still like to dial it?")
```

93

```
        if (!answer)
            continue;
    }
    NextContactToCall = contactDataObject[i];
    continue;
    }
}
```

IS_Event_NewPredictiveCall

Definition

This event is raised by Scripter when a predictive call is assigned to the agent by ACD. The event passes in the CallId of the call that is added to the users queue. The IS_Event_QueueObjectAdded event is also raised along side this event, since a call object has been added to the user's queue. This event is usually set up in the waiting for call page, to either listen for a predictive call or a preview call, and then redirect the agent to the appropriate page. Its prototype is IS_EventNewPredictiveCall(CallId).

Use the IS_Attr_Campaign attribute to identify the name of the campaign If names/values are required as a dynamic list, use IS_Event_DataPop instead of IS_Event_NewPredictiveCall. If IS_EventNewPredictiveCall is not defined, IS_Event_DataPop will be called by default.

Attributes

CallId

> Call identifier of the new call on the user's queue.

Example

```html
<head>
    <meta name="IS_Attr_CallId">
    <script language=javascript>
        function IS_Event_NewPredictiveCall(CallId) {
            // navigate to the data display page
            location.href = "intro.html";
        }
    </script>
</head>
```

IS_Event_NewPreviewCall

Definition

This event is fired when a new preview call is placed in a queue. It is fired after IS_Action_PlacePreviewCall has been invoked by a script. Its prototype is IS_Event_NewPreviewCall(CallId). Use the IS_Attr_Campaign attribute to identify the name of the campaign. If IS_EventNewPreviewCall is not defined, IS_Event_DataPop will be called by default. It is raised by Scripter when a preview call is presented to the agent. Note that this event is not the data pop, but the event that is raised after the agent issued an IS_Action_PlacePreviewCall function call.

Attributes

CallId

> Call identifier of the new call on the user's queue.

Example

```
<head>
    <script language=javascript>
        function IS_Event_NewPreviewCall(CallId) {
            // navigate to the data display page
            location.href = "intro.html";
        }
    </script>
</head>
```

IS_Event_PreviewCallSkipped

Definition

This event is emitted when a preview call is successfully skipped. This event will be emitted regardless of whether the call is skipped through the skip action in a custom script or through the skip button in the optional command toolbar at the bottom of the script.

Attributes

None.

Example

In a custom script, the event can be handled as follows:

```
function IS_Event_PreviewCallSkipped()
{ console.warn("preview skip was pressed!"); }
```

IS_Event_PredictiveCallReleased

Definition

This event is generated when a call is disconnected, transferred, or stolen from the call queue. Its prototype is IS_Event_PredictiveCallReleased(CallId). It is generated by Scripter when a call is either disconnected or transferred from the user's queue. The CallId of the disconnected call is passed to this event. This event can be used as a way to make sure an agent dispositions a call. For example, if the call was disconnected, and there was no disposition, you can set up a timer to alert the user that the call was disconnected and they need to disposition the call.

Attributes

CallId

> CallId is a string that contains the id of the callObject that has been disconnected.

Example

```html
<head>
    <script language=javascript>
        function IS_Event_PredictiveCallReleased(CallId) {
            // this is not a page where an agent can transfer a call
            alert("The call has been disconnected, or stolen.");
        }
    </script>
</head>
```

IS_Event_PreviewDataPop

Definition

This event is called by Scripter in Preview dialing mode. It is raised by Scripter when a preview data pop is presented to the agent. This event occurs before the IS_Event_NewPreviewCall. IS_Event_PreviewDataPop provides notification that the client can display a customer record, before the call is placed. This allows the agent to review the client record before pushing a button to initiate the call. Its prototype is IS_Event_PreviewDataPop(Names, Values). If IS_EventPreviewDataPop is not defined, IS_Event_DataPop will be called by default.

This event presents a JavaScript array of names and values that are associated with the preview call. When running in preview mode, the data is presented to the agent first, then the agent determines whether or not to place the call.

Attributes

Names

   A JavaScript array of the Dialer attribute names.

Values

   A JavaScript array of the Dialer attribute values

Example 1

```html
<head>
    <script>
        function IS_Event_PreivewDataPop(Names, Values) {
            // do something here
            location.href = "PreviewPop.htm";
        }
    </script>
</head>
```

Example 2

The example below is a typical waiting for call page, that listens for Preview and Predictive calls. Typically this type of page redirects the agent to another page to display the data that is associated with the call. See Interaction Scripter Attributes for information about retrieving values from call data.

```html
<head>
    <script>
        function IS_Event_PreivewDataPop(Names, Values) {
            // do something here
            location.href = "PreviewPop.htm";
        }
    </script>
</head>
```

PreviewTimeout Events

PreviewTimeout events are Predictive Events that are specifically designed for use with Preview campaigns that display a preview countdown timer. Using a campaign property designed specifically for Preview campaigns and set up in Dialer Manager, you can enable and configure a preview countdown timer that displays on the screen and indicates how much time agents have to review the preview pop before the contact is automatically dialed.

When the preview countdown timer is enabled, Dialer will send the following events to InteractionScripter.Net, which will in turn send them to the script. You can then use these events in a custom script to trigger additional processes. (Keep in mind that a countdown timer is independent of the script, but is only used for Preview campaigns.) These events are distinctive because rather than receiving a list of parameters like other events, these events receive argument objects.

> **TIP:** PreviewTimeout Events only function when running a script on an Outbound Dialer Server. If you are running a script on a Manual Calling Server, which by default doesn't support the preview countdown timer feature, then the PreviewTimeout Events will not function. They can be present in a script, they just won't function because on a Manual Calling Server an agent must manually initiate a call by clicking a button.

Event	Definition
IS_Event_PreviewTimeoutStarted	This event is fired when the preview timer starts.
IS_Event_PreviewTimeoutStopped	This event is fired when the preview timer stops or is canceled.
IS_Event_PreviewTimeoutExpired	This event is fired when the preview timer expires.

IS_Event_PreviewTimeoutStarted

Definition

The IS_Event_PreviewTimeoutStarted event is a Predictive Event that is specifically designed for use with Preview campaigns that use a preview countdown timer. When a countdown timer is enabled for a Preview campaign, Dialer will send the IS_Event_PreviewTimeoutStarted event to a custom script when the countdown timer starts.

The IS_Event_PreviewTimeoutStarted event will only function when running a script on an Outbound Dialer Server. If you are running a script on a Manual Calling Server, which by default doesn't support the preview countdown timer feature, then the IS_Event_PreviewTimeoutStarted event will not function. The IS_Event_PreviewTimeoutStarted event can be present in a script, it just won't function because on a manual calling server an agent must manually initiate a call by clicking a button.)

Attributes

The IS_Event_PreviewTimeoutStarted event has two attributes that are passed in as argument objects.

InteractionId

This is a string that holds the Id of the preview call for which the timeout has started.

Timeout

This is a DateObject that indicates when the timeout will expire.

Example

```html
<head>
    <script language=javascript>
        function IS_Event_PreviewTimeoutStarted(args) {
            var id = args.InteractionId;
            var timeout = args.Timeout;
            // insert other code here as needed...
        }
    </script>
</head>
```

IS_Event_PreviewTimeoutStopped

Definition

The IS_Event_PreviewTimeoutStopped event is a Predictive Event that is specifically designed for use with Preview campaigns that use a preview countdown timer. When a countdown timer is enabled for a Preview campaign, Dialer will send the IS_Event_PreviewTimeoutStopped event to a custom script to indicate that the countdown timer has stopped or has been canceled.

The IS_Event_PreviewTimeoutStopped event will only function when running a script on an Outbound Dialer Server. If you are running a script on a Manual Calling Server, which by default doesn't support the preview countdown timer feature, then the IS_Event_PreviewTimeoutStopped event will not function. (The IS_Event_PreviewTimeoutStopped event can be present in a script, it just won't function because on a Manual Calling Server an agent must manually initiate a call by clicking a button.)

Attributes

IS_Event_PreviewTimeoutStopped has one attribute that is passed in as an argument object.

InteractionId

>	This is a string that holds the ID of the preview call for which the timeout has stopped.

Example

```html
<head>
    <script language=javascript>
        function IS_Event_PreviewTimeoutStopped(args) {
            var id = args.InteractionId;
            // insert other code here as needed...
        }
    </script>
</head>
```

IS_Event_PreviewTimeoutExpired

Definition

The IS_Event_PreviewTimeoutExpired event is a Predictive Event that is specifically designed for use with Preview campaigns that use a preview countdown timer. When a countdown timer is enabled for a Preview campaign, Dialer will send the IS_Event_PreviewTimeoutExpired event to a custom script to indicate that the time on the countdown timer has expired.

An IS_Event_PreviewTimeoutExpired event will only function when running a script on an Outbound Dialer Server. If you are running a script on a Manual Calling Server, which by default doesn't support the preview countdown timer feature, then the IS_Event_PreviewTimeoutExpired event will not function. The IS_Event_PreviewTimeoutExpired event can be present in a script, it just won't function because on a Manual Calling Server an agent must manually initiate a call by clicking a button.

Attributes

The IS_Event_PreviewTimeoutExpired event has two attributes that are passed in as argument objects.

InteractionId

> This is a string that holds the ID of the preview call for which the timeout has expired.

Cancel

> This is a Boolean flag that can be used to cancel the automatic placement of the preview call. Set it to true to cancel the automatic call.

Example

```html
<head>
    <script language=javascript>
        var cancelPreview = false;
        function IS_Event_PreviewTimeoutExpired(args) {
            var id = args.InteractionId
            if (cancelPreview)
                args.Cancel = true;
            // insert other code here as needed...
        }
    </script>
</head>
```

Interaction Scripter Attributes

Attributes are data items passed by actions to the CIC server. A Dialer attribute is data from a column in a database that is associated with a campaign. Every Dialer database column is automatically associated with a script object of the same name with IS_Attr_ prefixed. For example, the database column "address" is available as script attribute "IS_ATTR_ADDRESS".

If the attribute is first declared in the script, it will go back to the Dialer server during a call complete function, and it can be accessed from a handler. In the CIC environment, an attribute is a piece of information about an object (such as a telephone call) that travels with the object. An example might be the telephone number of the individual called during a campaign. The server passes attributes to the client application when a new call event occurs. The client passes attributes to the server when a call-complete action is performed.

- Predictive attributes are attributes that are normally used with either a Predictive, Preview or Power dialing campaigns. These attributes are not to be used in blended environments, for example in inbound pages loaded in scripter. The predictive base view for dialer is not loaded in an inbound page, thus these attributes would not return any values.
- System Services attributes are supplemental predictive attributes that retrieve information about a Dialer agent, such as the agent's name, or ID . You can also use system services to change an agent's status.
- Custom attributes are also supported. Scripter provides the ability to create any attribute within a custom script. These attributes can be references to the actual values in the call list or can be a newly created attribute declared in a meta tag within the pages loaded in scripter.

Scripter attributes are normally data of some sort that is associated with a call that is on the user's queue. These attributes normally consist of the data that came from the call list that the campaign was dialing from. Custom attributes can be also created for added flexibility. These variables stay persistent through out the life of the call while it is on the users queue. Because of this persistence, it is very good practice at the time of disposition to clear out all relevant attributes to reduce the chance that stale data might populate the newly arrived call. All basic scripter attributes are prefixed with IS_Attr_ . This prefix tells Scripter that this is a attribute that is must retrieve or maintain data through out the session. A few system attributes are read-only. Most read-only attributes start with IS_System_, but a few IS_Attr_ attributes are read only also.

Predictive Attributes

A Dialer attribute is data from a column in a database that is associated with a campaign. Every Dialer database column will automatically become associated with a script object of the same name with IS_Attr_ prefixed. For example, the database column "address" is available as script attribute "IS_ATTR_ADDRESS". If the attribute is first declared in the script, it will go back to the Dialer server during a call complete function, and it can be accessed from a handler.

Predictive attributes are normally used with either a Predictive, Preview or Power dialing campaigns. These attributes are not to be used in blended environments. For example, you cannot use predictive attributes in inbound pages loaded into Scripter, since the predictive base view for Dialer is not loaded in an inbound page, and consequently predictive attributes would not return any values.

Reserved Read-Only Attributes

Treat these attributes as read-only and never create a database column that conflicts with these names. These parameters are available to the script, but a script should never modify their values.

Attribute	Description
IS_Attr_Attempts	The number of times that the server has attempted to call the targeted party.
IS_Attr_CallId	The identifier of the current call.
IS_Attr_CampaignID	Globally unique identifier (GUID) of the campaign object.
IS_Attr_CampaignGroup	If the Advanced Campaign Management feature is in use, this attribute will contain the name of the active Campaign Group in a Campaign Sequence. If a campaign is using rule groups as an automation tool, this attribute will contain the name of the active rule group.
IS_Attr_CampaignName	The name of the current campaign object.
IS_Attr_ContactCampaignID	Identifies which campaign the record came from.
IS_Attr_DialingMode	The dialing mode for the current campaign.
IS_Attr_I3_RowID	Represents the row id of the record that is presented to the agent. This row id is a unique value in the call list, and is usually the primary key in the call list table.
IS_Attr_Schedphone	Assigns the number of a call to schedule.
IS_Attr_Status	This read-only attribute contains the status of the current call record.
IS_Attr_Zone	The value of zone column in the call list

Tips for Using Global Variables

All sub attributes of **IS_Attr** elements are global. These variables persist while the client is running. It is a good practice to clear global data at the beginning of each call. If more than one element is given the same IS_ATTR_* name, unpredictable results will occur.

```
// create some attributes
IS_Attr_CallData.foo = "Hello";
IS_Attr_CallData.bar = "World";
```

IS_Attr_Attempts

Definition

This attribute contains the number of times that the server has attempted to call the targeted party. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Usage

Read   Yes
Write   No

Example

This example displays the attempts attribute in an edit field.

```html
<body>
    <input name="IS_Attr_Attempts">
<body>
```

IS_Attr_CallId

Definition

This read-only attribute represents the CallID of the current call object that is on the users' queue. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Usage

Read   Yes
Write   No

Example

This example shows how to display the values of various attributes in a simple web page.

```html
<html>

<head>
    <title>Data Pop</title>
</head>
<meta name=IS_System_ClientStatus>
<meta name="IS_Attr_CallID">
<meta name="IS_Attr_Zone">
<meta name="IS_Attr_DialingMode">
<meta name='IS_Attr_Attempts">
<meta name="IS_Attr_CampaignName">
<meta name="IS_System_AgentID">

<script language="javascript">
    function InitliazePageValues() {
        tagAgentID.innerText = IS_System.AgentID.value;
        tagAgentStatus.innerText = IS_System_ClientStatus.value;
        tagCallID.innerText = IS_Attr_CallID.value;
        tagZone.innerText = IS_Attr_Zone.value;
```

```
            tagCampaignName.innerText = IS_Attr_CampaignName.value;
        switch (IS_Attr_DialingMode.value) {
            case 0:
                tagDialingMode.innerText = "Power / Predictive";
                break;
            case 1:
                tagDialingMode.innerText = "Preview Mode";
                break;
            case 2:
                tagDialingMode.innerText = "Place Preview";
                break;
            case 3:
                tagDialingMode.innerText = "Own Agent Callback";
                break;
            case 4:
                tagDialingMode.innerText = "Own Agent Callback Preview";
                break;
            case 5:
                tagDialingMode.innerText = "Own Agent Callback Place Preview";
                break;
            case 7:
                tagDialingMode.innerText = "Agentless";
                break;
            case 8:
                tagDialingMode.innerText = "Precise Dial";
                break;
        }
    }
</script>
<body>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagAgentID">[Agent ID]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagAgentStatus">[Agent Status]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagCallID">[Call  ID]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagZone">[ZONE]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagDialingMode">[Dialing Mode]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagCampaignName">[Campaign Name]</em></font>
</body>
</html>
```

IS_Attr_CampaignID

Definition

This read-only attribute contains the id of the currently running campaign. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from buttons.

Usage

Read   Yes
Write   No

Example

This is an example of a campaign edit field.

```html
<head>
    <meta name=IS_Attr_CampaignID>
    <script language="javascript">
        window.onload = InitTagValues;
        function InitTagValues() {
            tagCampaignName.innerText = IS_Attr_CampaignID.value;
        }
    </script>
</head>
<body>
    <p>The ID of this campaign is: <em id="tagCampaignId">[Campaign Id]</em></p>
</body>
```

IS_Attr_CampaignName

Definition

Returns the name of the campaign object.

Usage

Read   Yes
Write   No

Example

This example shows how to display the values of various attributes in a simple web page.

```html
<html>

<head>
    <title>Data Pop</title>
</head>
<meta name=IS_System_ClientStatus>
<meta name="IS_Attr_CallID">
<meta name="IS_Attr_Zone">
<meta name="IS_Attr_DialingMode">
<meta name="IS_Attr_Attempts">
<meta name="IS_Attr_CampaignName">
<meta name="IS_System_AgentID">
<script language="javascript">
```

```
    function InitliazePageValues() {

        tagAgentID.innerText = IS_System.AgentID.value;
        tagAgentStatus.innerText = IS_System_ClientStatus.value;
        tagCallID.innerText = IS_Attr_CallID.value;
        tagZone.innerText = IS_Attr_Zone.value;
        tagCampaignName.innerText = IS_Attr_CampaignName.value;
        switch (IS_Attr_DialingMode.value) {
            case 0:
                tagDialingMode.innerText = "Power/Predictive";
                break;
            case 1:
                tagDialingMode.innerText = "Preview Mode";
                break;
            case 2:
                tagDialingMode.innerText = "Place Preview";
                break;
            case 3:
                tagDialingMode.innerText = "Own Agent Callback";
                break;
            case 4:
                tagDialingMode.innerText = "Own Agent Callback Preview";
                break;
            case 5:
                tagDialingMode.innerText = "Own Agent Callback Place Preview";
                break;
            case 7:
                tagDialingMode.innerText = "Agentless";
                break;
            case 8:
                tagDialingMode.innerText = "Precise Dial";
                break;
        }
    }
</script>
<body>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagAgentID">[Agent ID]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagAgentStatus">[Agent Status]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagCallID">[Call  ID]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagZone">[ZONE]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagDialingMode">[Dialing Mode]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagCampaignName">[Campaign Name]</em></font>
</body>
</html>
```

IS_Attr_CampaignGroup

Definition

If a campaign is using rule groups as an automation tool, this attribute contains the name of the active rule group in the campaign that is currently running. If the Advanced Campaign Management feature is in use, this attribute contains the name of the active Campaign Group in the Campaign Sequence that is currently running.

Usage

Read Yes

Write No

Example

```html
<html>

<head>
    <title>Standard Campaign Form</title>
    <meta name="IS_Attr_CampaignGroup" />
    <script language="javascript">
        window.onload = InitTagValues;
        function InitTagValues() {
            tagCampaignGroup.innerText = IS_Attr_CampaignGroup.value;
        }
    </script>
</head>
<body>
    <p>The Active Group is of this campaign is: <em id="tagCampaignGroup">[Group]</em>
</body>
</html>
```

IS_Attr_ContactCampaignID

Definition

This attribute is deprecated but will still be populated with the ID of the campaign associated with the current Dialer call.

Usage

Read   Yes

Write   No

IS_Attr_DialinsgMode

Definition

This read-only attribute returns the dialing mode for the current campaign. It represents the dialing mode of the Interaction Dialer call that is presented to the agent. The possible values are:

0	Power/Predictive mode
1	Preview mode
2	Place Preview
3	Own Agent Callback
4	Own Agent Callback Preview
5	Own Agent Callback Place Preview
7	Agentless
8	Precise

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Usage

Read   Yes

Write   No

Example

This is an example of a dialing mode edit field.

```html
<head>
<script language="javascript">
window.onload = InitTagValues;
function InitTagValues() {
    tagDialingMode.innerText = IS_Attr_DialingMode.value;
}
</script>
</head><body>
    <p>The current dialing mode is: <em id="tagDialingMode">[Dialing Mode]</em></p>
</body>
```

IS_Attr_I3_RowID

Definition

This read-only attribute represents the row id of the record that is presented to the agent. This row id is a unique value in the call list, and is usually the primary key in the call list table. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from the button(s).

Usage

Read   Yes
Write   No

Example

```html
<head>
    <script language="javascript">
        window.onload = InitTagValues;
        function InitTagValues() {
            tagRowID.innerText = IS_Attr_RowID.value;
        }
    </script>
</head>
<body>
    <p>The RowID attribute is: <em id="tagRowID">[RowID]</em></p>
</body>
```

IS_Attr_Schedphone

Definition

This attribute can be assigned the telephone number of a call to schedule. When implemented in a script, add a meta tag (or other named element) called 'is_attr_schedphone' whose value will be assigned the scheduled phonenumber. When this attribute is present in the script, you must ensure that it gets initialized to "" for every data pop.

Example

```html
<head>
    <meta name="IS_attr_schedphone">
    <meta name="IS_Action_CallComplete">
    <script language="javascript">
        function doSchedule() {
            IS_attr_schedphone.value = "555-1448"
            IS_Action_CallComplete.wrapupcode = "Scheduled";
            IS_Action_CallComplete.agentid = "Dev_PD_User1";
            IS_Action_CallComplete.click();
        }
    </script>
</head>
```

IS_Attr_Status

Definition

This read-only attribute contains the status of the current call record. The status column in the Contact List indicates the overall status of a contact, not the callable status of individual numbers, which is maintained in the Phone Number Detail table. This

approach allows a DNC status to be maintained in the PND table for individual telephone numbers associated with a contact. Scripter agents typically see only S, R, or C. The possible values are:

A	The maximum retry attempts for busy, answering machine, no answer etc. have been reached for the record. It will never be called again.
C	A callable record.
D	"D" is assigned to deleted contacts who have asked to have their numbers removed from the Contact List. This indicator is not used with scheduled calls. A "D" can only come from an agent disposition, wherein the callee has been asked to be removed from the Contact List.
E	The contact is excluded from being dialed.
F	A record "flagged" for customer review because all of a contact's numbers are uncallable. For example, if all attempts to dial phone numbers for a contact fail with SIT wrap-up codes, then that contact is essentially uncallable until a new contact column is added or some of the existing numbers are changed. Rather than mark the contact as uncallable, it is flagged with "F" so that customers can change the status back to "C" after they have added a new contact column, or updated contact numbers.
I	"I" stands for In Process. The record selection process has selected this record for processing, and the record has been passed to an Outbound Dialer server. However, the record may not have been dialed yet, and it has not been dispositioned by an agent. When a campaign stops, its active contacts are reset in the ContactList (status changed from 'I' to 'C'), and the active campaign ID (i3_activecampaignid) is used by a stored procedure to ensure that only contacts associated with this campaign are cleared.
O	"O" stands for auto-scheduled call. When the system schedules a call back according to the defined auto-schedule settings, Status is set to O to indicate that a callback has been scheduled, but has not been attempted yet.
R	A record that was rescheduled because the designated agent was not logged in to take the call. If these are campaign wide calls, then if no agents are logged in, calls will be rescheduled with "R" in the status field. If 'ignore recycles' is checked, auto scheduled calls will be distinguished from agent scheduled campaign wide calls and own agent callbacks.
S	A scheduled call. When an agent schedules a call back, Status is set to S to indicate that a callback has been scheduled, but has not been attempted yet.
T	An auto-rescheduled call, meaning that the scheduled call was rescheduled by dialer, but not an agent. This status is assigned to a scheduled callback that failed to reach a contact, and the maximum number of callback attempts has not been reached. When it is, the status will be changed to "D". The call list status will appear as 'C' for auto-scheduled and rescheduled auto-scheduled calls if ignore recycles is not checked.
U	Unusable call record. This status flag designates that the contact will not be called again.

Usage

Read   Yes

Write   No

Example

```html
<head>
    <script language="javascript">
        window.onload = InitTagValues;
        function InitTagValues() {
            tagStatus.innerText = IS_Attr_Status.value;
        }
    </script>
</head>
<body>
    <p>The call record status is: <em id="tagStatus">[Status]</em></p>
</body>
```

IS_Attr_Zone

Definition

This must match a value in the ZoneSet associated with the campaign. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from buttons.

Usage

Read   Yes
Write   No

Example

This is an example of a time zone edit field.

```html
<body>
    <input name="IS_Attr_Zone">
</body>
```

System Services Attributes

System Service elements retrieve information about an agent, such as the agent's name, ID, or client status. System Services attributes are read-only.

Attribute	Description
IS_System_AgentID	Returns the User ID of the agent.
IS_System_AgentName	Returns the Exchange display name of the agent.
IS_System_ClientStatus	Returns the current availability of the client (Available, Out to Lunch, etc.), and the available status messages defined on the server.

IS_System_AgentID

Definition

IS_System_AgentID returns the ID of an agent. See also IS_System_AgentName, which returns the name of the agent. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from buttons.

Example

```html
<head>
    <meta name="IS_Action_CallComplete">
    <meta name="IS_System_AgentID">
    <script language=javascript>
        function OwnAgentCallback(callBackTime) {
            IS_Action_CallComplete.agentID = IS_System_AgentID.value;
            IS_Action_CallComplete.callBackTime = callBackTime;
            IS_Action_CallComplete.click();
        }
    </script>
</head>
<body>
    Callback Time <input id=CallbackTime> (mm/dd/yyyy hh:mm)
    <input type=button value="Call Back" onclick='OwnAgentCallback(CallbackTime.value)'>
</body>
```

IS_System_AgentName

Definition

IS_System_AgentName returns the name of the agent. See also IS_System_AgentID, which return the ID of an agent. Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from buttons.

Example

In the example below, IS_Attr_Name must be a column returned from a database.

```html
<body>
    <p>Hello, this is <em name="IS_System_AgentName"></em>.</p>
    <p>May I please speak with Mr./Mrs. <em name="IS_Attr_Name"></em>?</p>
</body>
```

IS_System_ClientStatus

Definition

IS_System_ClientStatus.list retrieves an array of status messages from the CIC Server. Status messages are defined in Interaction Administrator under System Configuration | Status Messages. The default status messages are:
- ACD – Agent Not Answering
- At a Training Session
- At Lunch
- Available
- Available, No ACD
- On Vacation
- Available, Remote

- Do Not Disturb
- Follow Up
- Gone Home
- In a Meeting
- Out of the Office
- Out of Town

In a default system configuration, **IS_System_ClientStatus**.**list** returns the following array elements:
- IS_System_ClientStatus.list[0] = ACD – Agent Not Answering
- IS_System_ClientStatus.list[1] = At a Training Session
- IS_System_ClientStatus.list[2] = At Lunch
- IS_System_ClientStatus.list[3] = Available
- IS_System_ClientStatus.list[4] = Available, No ACD
- IS_System_ClientStatus.list[5] = Available, Remote
- IS_System_ClientStatus.list[6] = Do Not Disturb
- IS_System_ClientStatus.list[7] = Follow Up
- IS_System_ClientStatus.list[8] = Gone Home
- IS_System_ClientStatus.list[9] = In a Meeting
- IS_System_ClientStatus.list[10] = On Vacation
- IS_System_ClientStatus.list[11] = Out of the Office
- IS_System_ClientStatus.list[12] = Out of Town

IS_System_ClientStatus.list.length returns the total number of status messages returned. In a default server configuration, IS_System_ClientStatus.list.length would return 13. Note that the array is zero-based.

IS_SYSTEM_ClientStatus.**value** allows you to retrieve the agent's status. For example, alert(IS_System_ClientStatus.value);.

Scripter will recognize click events from any HTML element whose name has an associated action documented in this API (e.g.: "IS_Action_CallComplete"). If the script needs to associate several buttons with the same action, then define the action using a meta element and call the click event on the meta element from buttons.

> Starting with version Interaction Scripter 3.0 Service Update 7, IS_System_ClientStatus.list now returns only accessible statuses. In earlier versions of Scripter it returned all statuses defined on the system. Access to statuses can be restricted by User or Workgroup, so Scripter now returns only those statuses that are available to the user.

Example

```html
<html>
<head>
    <meta name="IS_Action_ClientStatus">
    <meta name="IS_System_ClientStatus">
    <script language=javascript>
        window.onload = Init;
        function Init() {
            if (!IS_System_ClientStatus.list)
                return;
            var availableStatuses = IS_System_ClientStatus.list;
            for (i = 0; i < availableStatuses.length; i++) {
                // populate with all valid statuses
                StatusList.add(new Option(availableStatuses[i]));
                // select if it is the current status
                if (availableStatuses[i] == IS_System_ClientStatus.value) {
                    StatusList.selectedIndex = i;
                }
            }
        }
        function StatusChange() {
            // set the status attribute to the text of currently displayed item
            IS_Action_ClientStatus.statusId = StatusList.item(StatusList.selectedIndex).text;
            IS_Action_ClientStatus.click();
        }
    </script>
</head>
<body>
```

```
        <select id="StatusList" onchange="StatusChange();"></select>
</body>
</html>
```

Custom Scripter Attributes

Scripter can create custom attributes within a custom script. These attributes can be references to the actual values in the call list or can be a newly created attribute declared in a meta tag within the pages loaded in Scripter.

For example, suppose that the call list has a column called EmployeeSalary that stores employee salary totals. When the page is popped to the agent, you want to take the salary and calculate the raise based on the percentage stored in the column RaisePercentage. The newly calculated salary is to be stored in a newly created attribute within the page, but not persisted to the database. The example below demonstrates how this can be achieved.

Example

```
<html>

<head>
    <title>New Salary Page</title>
</head>
<meta name="IS_Attr_EmployeeSalary">
<meta name="IS_Attr_RaisePercentage">
<meta name="IS_Attr_NewSalary">

<script language="javascript">
    function CalculateRaise() {
        IS_Attr_NewSalary.value = ((IS_Attr_RaisePercentage.value / 100) * IS_Attr_EmployeeSalary.value) +
IS_Attr_EmployeeSalary.value;
        tagOriginalSalary.innerText = IS_Attr_EmployeeSalary.value;
        tagRaisePerc.innerText = IS_Attr_RaisePercentage.value;
        tagNewSalary.innerText = IS_Attr_NewSalary.value;
    }
</script>
<body>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagOriginalSalary">[Original Salary]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagRaisePerc">[Raise Percentage]</em></font>
    <font size=5 color=FFFFFF style="bold">
        <em id="tagNewSalary">[New Salary]</em></font>
</body>
</html>
```

Interaction Scripter Behaviors

Behaviors are like command line parameters and are used to change the way that Scripter behaves when running custom scripts. This means that the behaviors are limited to the custom scripts that implement them rather than applying globally.

- **Predictive Behaviors** are only applicable for custom scripts associated with a Predictive, Power or Preview campaign.

Predictive Behaviors

Interaction Scripter Predictive behaviors are only applicable for custom scripts associated with a Predictive, Power or Preview campaign.

Behavior	Definition
IS_Bhvr_SuppressToast	When added to a custom script, this behavior will suppress the toast pops that appear on the screen when a user is logged into or out of a campaign automatically by a rule or manually by an administrator. When this behavior is added to the head section of a custom script, it will prevent toast pops for any campaign that is using the custom script while allowing other campaigns using base scripts or other custom scripts to continue to display the log in and log off toast pops.

IS_Bhvr_SuppressToast

Definition

When added to a custom script, this behavior will suppress the toast pops that appear on the screen when a user is logged into or out of a campaign automatically by a rule or manually by an administrator. When this behavior is added to the head section of a custom script, it will prevent toast pops for any campaign that is using the custom script while allowing other campaigns using base scripts or other custom scripts to continue to display the log in and log off toast pops.

Attributes

None.

Example

```
<head>
    <meta name="IS_Bhvr_SuppressToast" />
</head>
```

scripter object

The **scripter** object adds call, chat, conference, queue, and user objects whose methods and properties are useful in blended call center environments:

- The CallObject class implements methods that interact with call objects. Methods supported by CallObject can dial, transfer, pickup, listen to, record, pause, or pick up calls. This makes it easy to query or set the properties of call objects, such as the CallID, or StateString.
- The campaign object encapsulates the Name, ID and Status of a Dialer campaign.
- The ChatObject class manipulates chat objects. Chat objects are very similar to call objects. The source examples for call objects can be adapted for chat objects. The methods and properties supported by scripter.chatObject are listed below. Click on a link for information concerning input parameters and return values.
- The ConferenceObject class manages conference calls. It allows you to begin a conference call, add calls to the conference, and disconnect parties from the conference. Properties of the ConferenceObject provide notification, handling, and the ability to enumerate objects in the conference.
- The Queue object connects to queues, and provides access to properties supported by queue objects. An additional queue object, scripter.myQueue is functionally equivalent to the Queue class, except that the current user's queue is preattached to the object. (scripter.queue can connect to any queue).
- The User object obtains information about a specific CIC user, such as a list of user, station, line, and workgroup queues that the user can view and modify, the user's status, logged in state, etc.
- The dialer object encapsulates properties of the agent's session with Dialer and properties such as which campaigns the agent is active in.

Methods

scripter.createCallObject	Creates a new CallObject.
scripter.createChatObject	Creates a new ChatObject.
scripter.createConferenceObject	Creates a new ConferenceObject.
scripter.createQueue	Creates a new Queue object.
scripter.createUser	Creates a new User object.
scripter.trace	Generates a trace log entry to aid in script debugging.

Properties

scripter.breakStatus	Returns the agent's break status (On Break, Break Pending, Not on Break).
scripter.callObject	Provides access to a static CallObject. It returns the current active call.
scripter.chatObject	Provides access to a static ChatObject. It returns the currently active chat object.
scripter.conferenceObject	Provides access to a static ConferenceObject.
scripter.myQueue	Provides access to a static Queue object preconnected to the logged-in agent's queue.
scripter.myUser	Provides access to a static User object referring to the logged-in agent.
scripter.notifierName	Returns the server name of the notifier the agent is connected to.
scripter.queue	Provides access to a static Queue object.
scripter.user	Provides access to a static User object.
scripter.dialer	This dialer object encapsulates properties of the agent's session with Dialer and properties such as which campaigns the agent is active in.

Example

The code snippet below demonstrates how scripter queue and call objects work together.

```javascript
scripter.myQueue.objectAddedHandler = ObjectAdded;
function ObjectAdded(ObjType, ObjId) {
    // we only want call objects (type 2)
    if (ObjType == 2) {
        scripter.callObject.id = ObjId;
        var campaign = scripter.callObject.getAttribute("IS_Attr_CampaignID");
        if ((campaign < > "") {

                // move this page to the top
                IS_Action_SelectPage.click();
                // stop listening for object added notifications
                scripter.myQueue.objectAddedHandler = null;

                // pop the data
                location.href = "pop.htm";

            } else {

                // we are not interested in this call
                scripter.callObject.id = -1;
            }
        }
    }
```

scripter.createCallObject Method

Definition

This method creates a new CallObject. You must first create a call object before dialing a number, or setting up a consult transfer, or conference.

Prototype

```
scripter.createCallObject([out,retval] CallObject)
```

Syntax

```
myObj=scripter.createCallObject();
```

Input Parameters

None.

Return Values

CallObject

> The new CallObject that was created.

Example

```html
<html>
<head>
    <title>Scripter Object</title>
    <script type="text/JavaScript">
        function MakeCall() { var CallObject = scripter.createCallObject(); CallObject.Dial(Phonenumber.value,
false); }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button onclick="MakeCall();" value="Dial"></td>
            <td>Phone Number<input name="Phonenumber" id="Phonenumber"></td>
        </tr>
    </table>
</body>
</html>
```

scripter.createChatObject Method

Definition

This method creates a new ChatObject. Use this method to keep track of multiple inbound chats. The Scripter chat object does not allow placing of outbound chats.

Syntax

```
var objChat = scripter.createChatObject();
```

Prototype

```
scripter.createChatObject([out,retval] ChatObject)
```

Input Parameters

None.

Return Values

ChatObject

> The new ChatObject that was created.

scripter.createConferenceObject Method

Definition

This method creates a new ConferenceObject. Once this object is created, you must call the Add method with Call Ids to add specific calls to the conference.

Syntax

```
var objConf = scripter.createConferenceObject();
```

Prototype

```
scripter.createConferenceObject([out,retval] ConferenceObject);
```

Input Parameters

None.

Return Values

ConferenceObject

> The new ConferenceObject that was created.

Example

```html
<html>
<head>
    <script language="javascript">
        function TestConference() {

            // Create the callobject
            var CallObj2 = scripter.createCallObject();

            // assign the id of the object to the current Campaign Call object
            CallObj2.id = IS_ATTR_CallID.value;
            // Create the call object to dial out
            var CallObj1 = scripter.createCallObject()

            // Dial third party number
            CallObj1.Dial("5551212", false);
            var doConnect = window.confirm("Add party to conference?");
            if (!doConnect) {
                // do not add the party to the conference
                CallObj1.Disconnect();
                // pick up scripter call
                CallObj2.PickUp();
                return;
            }
            // create conference object
            var ConfObj = scripter.createConferenceObject();

            // once call is added, conference object is valid
            CallObj1.pickup();
            ConfObj.Create(CallObj1);
            // add second call object
            CallObj2.pickup();
            ConfObj.add(CallObj2);
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button onclick="Conference();" value="Conference"></input></td>
            <td>Phone 1</td>
            <td><input name="Phone1" id="Phone1"></input></td>
        </tr>
    </table>
</body>
</html>
```

scripter.createQueue Method

Definition

This method creates a new [Queue](#) object.

Syntax

```
var objQueue = scripter.createQueue();
```

Prototype

```
scripter.createQueue([out,retval] Queue)
```

Input Parameters

None.

Return Values

Queue

> The new [Queue](#) object that was created.

scripter.createUser Method

Definition

This method creates a new [User Object](#).

Syntax

```
var objUser = scripter.createUser();
```

Prototype

```
scripter.createUser([out,retval] User)
```

Input Parameters

None.

Return Values

User

> The new [User Object](#) that was created.

scripter.trace Method

Definition

This method works like [IS_Action_Trace](#), by generating an entry to the trace log to aid in script debugging.

Syntax

```
scripter.trace(message, level);
```

Prototype

```
scripter.trace([in] String message, [in, optional] Long level)
```

Input Parameters

Message

       Message is the trace message and is a string.

Level

       Level is the tracing level and must be one of the following:

0	Error
1	Warning
2	Status
3	Note

       If level is invalid or missing, "Status" is used instead.

Return Values

None.

scripter.breakStatus Property

Definition

This property returns the agent's break status (On Break, Break Pending, Not on Break).

Syntax

```
scripter.breakStatus
```

Usage

Read   Yes
Write   No

Value Assigned

None.

Value Returned

breakStatus

NotOnBreak	0
BreakPending	1
OnBreak	2

scripter.callObject Property

Definition

This property provides access to a static CallObject. See the scripter.createConferenceObject method for sample code. Although scripter.callObject provides access to a callObject, it isn't necessarily the *active* callObject. Developers must set the CallID property.

Syntax

```
scripter.callObject
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

CallObject

> Returns the same instance of the CallObject each time.

scripter.chatObject Property

Definition

This convenience property provides access to a static [ChatObject](). Although scripter.chatObject provides access to a chatObject, it isn't necessarily the *active* chatObject. Use the chatObject.id property to determine if it is a valid chat.

Syntax

```
scripter.chatObject
```

Usage

Read   Yes
Write   No

Value Assigned

None.

Value Returned

ChatObject

> Returns the same instance of the [ChatObject]() each time.

scripter.conferenceObject Property

Definition

This property provides access to a static ConferenceObject. Use the scripter.conferenceObject.id property to determine if a conference exists. A user can generate only one conference Id at a time. See scripter.createConferenceObject method for sample code.

Syntax

```
scripter.conferenceObject
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

ConferenceObject

> Returns the same instance of the ConferenceObject each time.

scripter.myQueue Property

Definition

This property provides access to a static Queue object preconnected to the logged-in agent's queue.

Syntax

```
scripter.myQueue
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Queue

> Returns the same instance of the Queue object each time.

scripter.myUser Property

Definition

This property provides access to a static User object referring to the logged-in agent.

Syntax

`scripter.myUser`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

User

>   Returns the same instance of the User object each time.

Scripter.notifierName Property

Definition

This property returns the machine name of the server that the agent is currently logged into.

Syntax

`scripter.notifierName`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

notifierName

>   The name of the Notifier the agent is connected to.

scripter.queue Property

Definition

This property provides access to a static <u>Queue</u> object. This object remains the same throughout all scripts and pages. The name and type must be set to retrieve other properties.

Syntax

```
scripter.queue
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Queue

>Returns the same instance of the <u>Queue</u> object each time.

scripter.user Property

Definition

This property provides access to a static <u>User</u> object. You must first set the Id to retrieve other properties of this object.

Syntax

```
scripter.user
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

User

>Returns the same instance of the <u>User</u> object each time.

CallObject

CallObject manipulates call objects (telephone calls). This object supports standard telephony operations, such as placing, muting, recording, or disconnecting a call. In addition, the CallObject supports specialized operations normally performed by the CIC client. For example, CallObject methods can be used to make a call private, or send a call to voice mail. The methods and properties supported by the CallObject are listed below. Click on a link for information concerning input parameters and return values.

Methods

CallObject.blindTransfer	Performs a blind transfer to the specified telephone number. Use a blind transfer if you do not need to speak with the recipient before transferring a call. If the intended recipient does not answer, the call is sent to the recipient's voice mail.
CallObject.consultTransfer	Performs a consult transfer. Use a consult transfer if you need to speak with the recipient before transferring the call. If the intended recipient does not answer the phone, you can resume your conversation with the caller, transfer the call to the intended recipient's voice mail, or transfer the call to another person.
CallObject.currentDialerCallId	Returns the Id of the current active dialer interaction. This can be used to set a CallObject's id to initialize it with the current active dialer call.
CallObject.dial	Dials a phone number. If you want to catch errors, use a Handler or Object Watcher to receive error codes. Using Dial is the same as calling ExtendedDial with TimeoutSecs=15 and no call analysis.
CallObject.disconnect	Disconnects the current call.
CallObject.extendedDial	Dials a number, allows for call analysis, and optionally forces a timeout within a specified timeout period.
CallObject.getAttribute	Retrieves the value of the specified call object attribute.
CallObject.hold	Places a call on hold.
CallObject.listen	Allows a CIC user to listen in on a call.
CallObject.mute	Mutes a call so the remote party cannot hear what the local (CIC) party is saying.
CallObject.pause	Pauses recording of the current call.
CallObject.pickup	Picks up (answers) a call.
CallObject.playDigits	Plays DTMF tones for the string of digits provided as an input parameter.
CallObject.private	Makes a call private so it cannot be listened to or recorded by another CIC user.
CallObject.record	Records a call.
CallObject.setAttribute	Sets the value of the specified call object attribute.
CallObject.voicemail	Sends a call to the current user's voice mail.
CallObject.pauseSecureRecord	Can be used to avoid recording sensitive information, such as a Social Security number or credit card number, when connected to a call interaction.
CallObject.resumeSecureRecord	Resumes recording that was paused by invoking the CallObject.pauseSecureRecord method.

Callbacks

CallObject.errorHandler	Is invoked by when an internal error occurs in the call object. If you pass the name of a user-defined function to CallObject.errorHandler, the function will be called when this event occurs. This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.
CallObject.stateChangeHandler	Is invoked whenever the call state changes. This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.
CallObject.callObjectInitializedHandler	Allows a script to dial a number after waiting asynchronously for a call object to be created. Use this callback only in scripts for Interaction Connect.

Properties

CallObject.conferenceId	Returns a conference object ID if the call is included in a conference call.
CallObject.creationTime	Returns the date and time that the call object was created.
CallObject.direction	Indicates the direction of the call (e.g. inbound, outbound, etc.).
CallObject.id	Returns or sets the unique identifier of a call object.
CallObject.isHeld	Indicates whether or not a call is on hold.
CallObject.isMonitored	Indicates whether or not a call is being listened to.
CallObject.isMuted	Indicates whether or not a call is muted.
CallObject.isParty	Indicates whether the call is included in a conference call.
CallObject.isPaused	Indicates whether recording of a call has been paused.
CallObject.isPrivate	Indicates whether is in a private state, meaning that no one can listen in on (monitor) the call.
CallObject.isRecording	Indicates whether a call is being recorded.
CallObject.lastError	Retrieves the text of the last error that occurred in the CallObject. Each time a method or property is called on the CallObject, this value is cleared.
CallObject.lastErrorId	Retrieves the id of the last error that occurred in the CallObject. Each time a method or property is called on the CallObject, this value is cleared.
CallObject.localId	Returns the Station Id associated with the call.
CallObject.localLocation	Returns the phone extension of the station participating in an inbound or outbound call.
CallObject.localName	Returns the name of the logged in user.
CallObject.remoteId	Returns the formatted telephone number of the person outside CIC who is making or receiving a call.
CallObject.remoteLocation	Returns the unformatted telephone number of the person outside CIC who is making or receiving a call.
CallObject.remoteName	Returns or sets the name of the caller.
CallObject.state	Returns the integer value corresponding to the call state of the call object.
CallObject.stateString	Retrieves or sets the string value that is displayed as call state information in the CIC client.

CallObject.blindTransfer Method

Definition

This method performs a blind transfer to the specified telephone number. Use a blind transfer if you do not need to speak with the recipient before transferring a call. If the intended recipient does not answer, the call is sent to the recipient's voice mail.

Syntax

```
CallObject.blindTransfer(ToNumber);
```

Prototype

```
CallObject.blindTransfer(
    [in] String ToNumber
)
```

Input Parameters

ToNumber

>Telephone number of the transfer destination.

Return Values

None.

Example

```html
<head>
    <script language="javascript">
        function blindTransfer() {
            CallObj = scripter.callObject;
            CallObj.blindTransfer(Phone1.value);
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button value="B. Transfer" onclick="blindTransfer();"></input>
            </td>
            <td>Number</td>
            <td><input name="Phone1" id="Phone1"></input>
            </td>
        </tr>
    </table>
</body>
```

CallObject.callObjectInitializedHandler Callback

Definition

This callback allows a script to dial a number after waiting asynchronously for a call object to be created.

Compatibility

Use this callback only in scripts for Interaction Connect.

Example

```
function main()
{
  var callObject = scripter.createCallObject();
  callObject.callObjectInitializedHandler = uponCallObjectInitialization;
  callObject.dial('3174566324');
}
function uponCallObjectInitialization(callObject)
{
  console.log('printing the call direction ' + callObject.direction);
}
```

CallObject.consultTransfer Method

Definition

Use this method to perform a consult transfer if you need to speak with the recipient before transferring the call. If the intended recipient does not answer the phone, you can resume your conversation with the caller, transfer the call to the intended recipient's voice mail, or transfer the call to another person.

Syntax

```
CallObject.consultTransfer(pVal)
```

Prototype

```
CallObject.consultTransfer(
  [in] String WithCallID
)
```

Input Parameters

WithCallID

> The Call ID to transfer this call to.

Return Values

None.

Example

```javascript
//* @param remoteNumber is the number to dial for the third party
function ConsultTransfer() {
    // call the remote number
    var CallObject2 = scripter.CreateCallObject();
    scripter.callObject.id = IS_Attr_CallId.value;
    CallObject2.dial(RemoteNumber, false);
    var doConnect = window.confirm("Transfer call?");
    if (doConnect) //the agent selected "OK" in the confirmation box
    {
        // make the transfer
        scripter.callObject.consultTransfer(CallObject2.id);
        scripter.callObject.id = -1; //release the callObject
        location.href = "inboundindex.htm";
    }
}
```

CallObject.currentDialerCallId Method

Definition

Returns the Id of the current active dialer interaction. This can be used to set a CallObject's id to initialize it with the current active dialer call.

Example

```
function makeCurrentCallObject()
{ currentCallObject = scripter.createCallObject(); currentCallObject.callObjectInitializedHandler =
scripterCallInitialized; currentCallObject.id = currentCallObject.currentDialerCallId(); }
```

Value Returned

Interaction id

 The id of the current interaction, in string form.

CallObject.dial Method

Definition

This method dials a phone number. If you want to catch errors, use a Handler or Object Watcher to receive error codes. Using Dial is the same as calling ExtendedDial with TimeoutSecs=15 and no call analysis. After calling a CallObject's Dial function, the objectSpecificChangeHandler will be called to give an event code describing the outcome of the dial operation.

Syntax

```
CallObject.dial(Number, CallHandlerOnSuccess);
```

Prototype

```
CallObject.dial(
  [in] String Number,
  [in] Boolean CallHandlerOnSuccess
)
```

Input Parameters

Number

 The telephone number to call.

CallHandlerOnSuccess

 A Boolean value. Set True to call the event handler when the call is connected.

Return Values

None.

Example

See the scripter.createConferenceObject method for sample code.

CallObject.disconnect Method

Definition

This method disconnects the current call.

Syntax

```
CallObject.disconnect();
```

Prototype

```
CallObject.disconnect()
```

Input Parameters

None.

Return Values

None.

Example

See the scripter.createConferenceObject method for sample code.

CallObject.extendedDial Method

Definition

This method dials a number, allows for call analysis, and optionally forces a timeout within a specified timeout period.

Syntax

```
CallObject.extendedDial(Number, TimeoutSecs, CallAnalysis, CallHandlerOnSuccess);
```

Prototype

```
CallObject.extendedDial(
  [in] string Number,
  [in] short TimeOutSecs,
  [in] Boolean CallAnalysis,
  [in] Boolean CallHandlerOnSuccess // not used, always set to True
)
```

Input Parameters

Number

    A string containing the telephone number to call.

TimeoutSecs

A short value indicating the maximum number of seconds to wait for an answer. Fifteen seconds is typically a good value to specify for this parameter.

CallAnalysis

Set this Boolean value True if the call should be analyzed for special error conditions, such as Operator interrupt, Busy Signal, etc.

CallHandlerOnSuccess

This parameter should always be set True To determine the state of the call after it has been dialed, please use myQueue.objectChangedHandler as shown in the example below. Keep in mind that neither the TimeOutSecs parameter nor the CallHandlerOnSuccess parameter of the CallObject.extendedDial Method are supported in IceLib.

Return Values

None.

Example

```
var CallIDentifier = 0;
function MakeCall() {
    var CallObject = scripter.createCallObject();
    CallObject.extendedDial(Phonenumber.value, 30, "TRUE", "TRUE");
    CallIDentifier = CallObject.id;
    scripter.myQueue.objectChangedHandler = CheckCallID;
}
function CheckCallID(TypeId, ObjectId) {
    if (ObjectId == CallIDentifier) {
        var CallObject = scripter.callObject;
        CallObject.id = ObjectId;
        if (CallObject.state == 105) {
            alert("call connected successfully");
            scripter.myQueue.objectChangedHandler = null;
            CallIDentifier = 0;
        }
    }
}
```

CallObject.getAttribute Method

Definition

This method retrieves the value of the specified call object attribute. To assign a value, use the CallObject.setAttribute method.

Syntax

```
CallObject.getAttribute(Name);
```

Prototype

```
CallObject.getAttribute(
  [in] string Name
  [out] string Value
)
```

Input Parameters

Name

The name of a call object attribute (e.g., CallID, StationName, Language, etc.) or a custom call attribute.

Return Values

Value

The value of the requested attribute is returned as a string value (e.g., 1078924, KevinKPC, Spanish).

Example

```javascript
scripter.myQueue.objectAddedHandler = ObjectAdded;
function ObjectAdded(ObjType, ObjId) {
    // we only want call objects (type 2)
    if (ObjType == 2) {
        scripter.callObject.id = ObjId;
        var campaign = scripter.callObject.getAttribute("IS_Attr_CampaignID");
        var workgroup = scripter.callObject.getAttribute("AssignedWorkgroup");
        if ((campaign < > "")) {
            // move this page to the top
            IS_Action_SelectPage.click();
            // stop listening for object added notifications
            scripter.myQueue.objectAddedHandler = null;
            // pop the data
            location.href = "pop.htm";
        } else {
            // we are not interested in this call
            scripter.callObject.id = -1;
        }
    }
}
```

CallObject.hold Method

Definition

This method places a call on hold.

Syntax

```
CallObject.hold();
```

Prototype

```
CallObject.hold()
```

Input Parameters

None.

Return Values

None.

Example

```javascript
function Hold() {
    var CallObject = scripter.callObject;
    if (CallObject.isHeld) {
        CallObject.pickup();
    } else {
        CallObject.hold();
    }
    CallObject = null;
}
```

CallObject.listen Method

Definition

This method allows a CIC user to listen in on a call. This method supports two optional parameters that identify the name of the queue and the queue type. If this method is called without parameters, the logged-in user's queue is used by default. To listen to a different queue, you must specify QueueName and QueueType parameters.

Syntax

```
CallObject.listen(QueueName, QueueType);
```

Prototype

```
CallObject.listen(
  [in] string QueueName,
  [in] int QueueType
)
```

Input Parameters

QueueName

> The name of the queue that you wish to connect to the queue object.

Type

> Type is an integer representing a queue type. QueueType is optional and defaults to 9 (User Queue). Valid values for queue types are:

3	Station queue
9	User queue
10	Workgroup queue
15	Line queue

Return Values

None.

CallObject.mute Method

Definition

This method mutes a call so the remote party cannot hear what the local (CIC) party is saying. This method works as a toggle to turn call muting on or off. Call the method a second time to unmute the call.

Syntax

```
CallObject.mute();
```

Prototype

```
CallObject.mute()
```

Input Parameters

None.

Return Values

None.

Example

```javascript
function Mute() {
    var CallObject = scripter.callObject;
    CallObject.mute();
    if (CallObject.isMuted) {
        alert("Call has been Muted");
    }
    CallObject = null;
}
```

CallObject.pause Method

Definition

This method pauses recording of the current call. This method performs a toggling action. To resume recording, call the method again. To stop recording, use CallObject.record.

Syntax

```
CallObject.pause();
```

Prototype

```
CallObject.pause();
```

Input Parameters

None.

Return Values

None.

Example

```javascript
function Record() {
    var CallObject = scripter.callObject;
    CallObject.record();
    CallObject = null;
}
function Pause() {
    var CallObject = scripter.callObject;
    if (CallObject.isRecording || CallObject.isPaused) {
        CallObject.pause();
    }
    CallObject = null;
}
```

CallObject.pickup Method

Definition

This method picks up (answers) a call. See the CallObject.hold method for example code.

Syntax

```
CallObject.pickup();
```

Prototype

```
CallObject.pickup()
```

Input Parameters

None.

Return Values

None.

CallObject.playDigits Method

Definition

This method plays DTMF tones for the string of digits provided as an input parameter. DTMF stands for Dual Tone Multi-Frequency. This term describes the tones generated when buttons are pressed on a touch tone telephone. Each tone is actually a combination of two tones, one high frequency, and one low frequency.

Syntax

```
CallObject.playDigits(StringOfDigits);
```

Prototype

```
CallObject.playDigits(
  [in] string StringOfDigits
)
```

Input Parameters

StringOfDigits

A string of digits to play. The string passed must contain only numeric digits, and may not contain alphabetic characters.

Return Values

None.

CallObject.private Method

Definition

Call this method to make a call private so it cannot be listened to or recorded by another CIC user. This method toggles privacy mode on or off. Call the method a second time to toggle privacy mode off.

Syntax

```
CallObject.private();
```

Prototype

```
CallObject.private()
```

Input Parameters

None.

Return Values

None.

Example

```javascript
function Private() {
    var CallObject = scripter.callObject;
    CallObject.private();
    If(CallObject.isPrivate) {
        alert("Call is now Private");
    }
}
```

CallObject.record Method

Definition

This method records a call. To stop recording, call the method a second time. To pause recording, use the CallObject.pause method. See CallObject.pause method for example code.

Syntax

```
CallObject.record();
```

Prototype

```
CallObject.record();
```

Input Parameters

None.

Return Values

None.

CallObject.setAttribute Method

Definition

This method sets the value of the specified call object attribute. To retrieve a value, use the [CallObject.getAttribute method](#).

Syntax

CallObject.setAttribute(Name, Value);

Prototype

```
CallObject.setAttribute(
    [in] string Name,
    [in] string Value
)
```

Input Parameters

Name

> The name of a call object attribute& (e.g., CallID, StationName, Language, etc.), or the name of a custom (user-defined) call attribute.

Value

> The value that will be assigned to the specified call object attribute (e.g. 1078924, KevinKPC, Spanish).

Return Values

None.

CallObject.voicemail Method

Definition

This method sends a call to the current user's voice mail.

Syntax

```
CallObject.voicemail();
```

Prototype

```
CallObject.voicemail()
```

Input Parameters

None.

Return Values

None.

Example

```
function Voicemail() {
    var CallObject = scripter.callObject;
    CallObject.voicemail();
}
```

CallObject.errorHandler Callback

Definition

CallObject.errorHandler is invoked by when an internal error occurs in the call object. If you pass the name of a user-defined function to CallObject.errorHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
CallObject.errorHandler (FunctionName);
```

Input Parameter

FunctionName

>   The name of the function to call when an error occurs.

Example

```
<script language="JavaScript">
    window.onload = Init;
    function Init() {
        scripter.callObject.errorHandler = ErrorHandler;
    }
    function ErrorHandler(ErrorId, ErrorText) {
        alert("Error occured.\n\nError Id: " + ErrorId + "\nError Text: " + ErrorText);
    }
</script>
```

CallObject.stateChangeHandler Callback Property

Definition

This method is called whenever this object's state changes. If you pass the name of a user-defined function to CallObject.stateChangeHandler, the function will be called whenever the call state changes.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
CallObject.stateChangeHandler(StateId, StateString)
```

Usage

Read   Yes

Write   Yes

Return Values

StateId

StateID is a number that represents the new call state of the object being watched.

Call State	String Value
Alerting	1
Connected	105
Dialing	103
Disconnected	106
Initializing	100
ManualDialing	102
Offering	101
OnHold	6
Proceeding	104
StationAudio	107

StateString

StateString is a string that describes the call state:

StateString	Description
Initalizing	CIC is formatting the telephone number and looking for a line on which to place the outbound call. This state applies to inbound and outbound calls.
Offering	The call has been placed in a queue, but the call is not alerting. CIC is determining if the called party is available to take the call. This state applies to inbound calls only.
Dialing	CIC is dialing the remote telephone number. This state applies to outbound calls only.
Proceeding	The call is proceeding through the outside telephone network. 'Proceeding' is used if a CIC client user has enabled Call Analysis. This state applies to outbound calls only.
Connected	Both parties are connected and are able to speak with each other. This state applies to inbound and outbound calls.
On Hold	The call is on hold. This state applies to inbound and outbound calls.
Disconnected	The call is no longer active. This state applies to inbound and outbound calls.
Manual Dialing	A telephone handset has been picked up and a dial tone is being generated. This state applies to outbound calls.
Station Audio	An audio clip is being played to one or more CIC client users.
Alerting	A CIC client user is being notified that he or she has an incoming call. This state applies to inbound calls.

Variables

Function Pointer

A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo".

When defining your custom function, you should declare StateID and StateString as arguments. e.g.: function foo(StateID, StateString).

CallObject.conferenceId Property

Definition

This property returns the conference object ID if the call is included in a conference call. Use CallObject.isParty to determine whether the call is part of a conference call.

Syntax

`CallObject.conferenceId`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

ConferenceID

> The value returned is a number identifying the conference object. If the call is part of a conference call, the conference object ID is returned. If the call is not part of a conference call, −1 is returned.

CallObject.creationTime Property

Definition

This property returns the date and time that the call object was created.

Syntax

`CallObject.creationTime`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Date

> The date and time are passed as a script Date object.

CallObject.direction Property

Definition

This property allows you to query the direction of the call (e.g. inbound, outbound, etc.).

Syntax

`CallObject.direction`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Direction

> The integer return value represents the call direction (0 = inbound, 1 = outbound, 2 = indeterminate direction or call is in a manual dialing state).

CallObject.id Property

Definition

This property returns or sets the unique identifier of a call object. A call identifier is typically a number composed of 10 digits. See CallObject.extendedDial method for a source code example.

Syntax

`CallObject.id`

Usage

Read   Yes

Write   Yes

Value Assigned

CallID

> To set the ID property, assign a unique number that identifies the call object.

Value Returned

CallID

> The value returned is a number identifying the call object.

CallObject.isHeld Property

Definition

This property indicates whether or not a call is on hold.

Syntax

`CallObject.isHeld`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Boolean

     If the call is currently on hold, returns True. Otherwise, returns False.

Example

```
function Hold() {
    var CallObject = scripter.callObject;
    if (CallObject.isHeld) {
        CallObject.pickup();
    } else {
        CallObject.hold();
    }
    CallObject = null;
}
```

CallObject.isMonitored Property

Definition

This property indicates whether or not a call is being listened to.

Syntax

`CallObject.isMonitored`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Boolean

     Returns True if the call is being listened to. Otherwise, returns False.

Example

```javascript
function Monitored() {
    var CallObject = scripter.callObject;
    if (CallObject.isMonitored) {
        alert("Call has been monitored.");
    }
}
```

CallObject.isMuted Property

Definition

This property indicates whether or not a call is muted.

Syntax

`CallObject.isMuted`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Boolean

> If the call is currently muted, returns True. Otherwise, returns False.

Example

```javascript
function Mute() {
    var CallObject = scripter.callObject;
    CallObject.mute();
    if (CallObject.isMuted) {
        alert("Call has been Muted");
    }
    CallObject = null;
}
```

CallObject.isParty Property

Definition

This property indicates whether the call is included in a conference call. Use CallObject.conferenceId to retrieve the conference ID number.

Syntax

```
CallObject.isParty
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Boolean

   If the call is included in a conference call, returns True. Otherwise, returns False.

CallObject.isPaused Property

Definition

This property indicates whether recording of a call has been paused.

Syntax

```
CallObject.isPaused
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Boolean

      If recording has been paused, returns True. Otherwise, returns False.

Example

```javascript
function Record() {
    var CallObject = scripter.callObject;
    CallObject.record();
    CallObject = null;
}
function Pause() {
    var CallObject = scripter.callObject;
    if (CallObject.isRecording || CallObject.isPaused) {
        CallObject.pause();
    }
    CallObject = null;
}
```

CallObject.isPrivate Property

Definition

This property indicates whether the call is in a private state, meaning that no one can listen in on (monitor) the call.

Syntax

```
CallObject.isPrivate
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Boolean

The property returns True if the call is in a private state; otherwise, returns False.

Example

```
function Private() {
    var CallObject = scripter.callObject;
    CallObject.private();
    If(CallObject.isPrivate) {
        alert("Call is now Private");
    }
}
```

CallObject.isRecording Property

Definition

This property indicates whether a call is being recorded.

Syntax

```
CallObject.isRecording
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Boolean

     Returns True if the call is being recorded; otherwise, returns False.

Example

```javascript
function Record() {
    var CallObject = scripter.callObject;
    CallObject.record();
    CallObject = null;
}
function Pause() {
    var CallObject = scripter.callObject;
    if (CallObject.isRecording || CallObject.isPaused) {
        CallObject.pause();
    }
    CallObject = null;
}
```

CallObject.lastError Property

Definition

This property retrieves the text of the last error that occurred in the CallObject. Each time a method or property is called on the CallObject, this value is cleared.

Syntax

`CallObject.lastError`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

String

> The text message of the last error that occurred.

Example

```
function GetLastError() {
    alert(scripter.callObject.lastError);
}
```

CallObject.lastErrorId Property

Definition

This property retrieves the id of the last error that occurred in the CallObject. Each time a method or property is called on the CallObject, this value is cleared.

Syntax

```
CallObject.lastErrorId
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

Integer

       The id of the last error that occurred.

Example

```html
<html>
<head>
    <script language=javascript>
        window.onload = Init();
        function Init() {
            setTimer("SetErrorInfo();", 1000);
        }
        function SetErrorInfo() {
            ErrorInfo.value = scripter.callObject.lastError;
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td colspan=4><input name="ErrorInfo" value="No Errors" style="width:600"></input>
            </td>
        </tr>
    </table>
</body>
</html>
```

CallObject.localId Property

Definition

This property returns the Station Id associated with the call.

Syntax

```
CallObject.localId
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

localID

>   When you retrieve this value, a string identifying the Station Id is returned.

Example

```html
<html>

<head>
    <TITLE>Scripter Object</TITLE>

    <meta name="IS_System_AgentName">

    <script language="JavaScript">
        var CallIDentifier = 0;
        function MakeCall() {
            var CallObject = scripter.createCallObject();
            CallObject.extendedDial(Phonenumber.value, 30, "TRUE", "TRUE");
            CallIDentifier = CallObject.id;
            scripter.myQueue.objectChangedHandler = CheckCallID;
            LocalId.value = CallObject.localId;
            LocalLocation.value = CallObject.localLocation;
            LocalName.value = CallObject.localName;
        }
        function CheckCallID(TypeId, ObjectId) {

            if (ObjectId == CallIDentifier) {
                var CallObject = scripter.callObject;
                CallObject.id = ObjectId;
                if (CallObject.state == 105) {
                    alert("call connected successfully");
                    RemoteId.value = CallObject.remoteId;
                    RemoteLocation.value = CallObject.remoteLocation;
                    RemoteName.value = CallObject.remoteName;
                    scripter.myQueue.objectChangedHandler = null;
                    call.value = 0;
                    CallIDentifier = 0;
                }
            }
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button value="dial" onclick="MakeCall();"></input>
            </td>
            <td>Phonenumber</td>
            <td><input name="Phonenumber" id="Phonenumber"></input>
            </td>
        </tr>
        <tr>
```

```html
        <td>Local Id</td>
        <td><input name="LocalId" value="" style="width:100"></input>
        </td>
        <td>Local Location</td>
        <td><input name="LocalLocation" value="" style="width:100"></input>
        </td>
        <td>Local Name</td>
        <td><input name="LocalName" value="" style="width:100"></input>
        </td>
    </tr>
    <tr>
        <td>Remote Id</td>
        <td><input name="RemoteId" value="" style="width:100"></input>
        </td>
        <td>Remote Location</td>
        <td><input name="RemoteLocation" value="" style="width:100"></input>
        </td>
        <td>Remote Name</td>
        <td><input name="RemoteName" value="" style="width:100"></input>
        </td>
    </tr>
    </table>
</body>
</html>
```

CallObject.localLocation Property

Definition

This property returns the phone extension of the station participating in an inbound or outbound call.

Syntax

```
CallObject.localLocation
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

localLocation

>   The phone extension of the station participating in an inbound or outbound call.

Example

```html
<html>
<head>
    <title>Scripter Object</title>
    <meta name="IS_System_AgentName">
    <script language="JavaScript">

        var CallIDentifier = 0;
        function MakeCall() {
            var CallObject = scripter.createCallObject();
```

```
                CallObject.extendedDial(Phonenumber.value, 30, "TRUE", "TRUE");
                CallIDentifier = CallObject.id;
                scripter.myQueue.objectChangedHandler = CheckCallID;
                LocalId.value = CallObject.localId;
                LocalLocation.value = CallObject.localLocation;
                LocalName.value = CallObject.localName;
        }           function CheckCallID(TypeId, ObjectId) {

            if (ObjectId == CallIDentifier) {
                var CallObject = scripter.callObject;
                CallObject.id = ObjectId;
                if (CallObject.state == 105) {
                    alert("call connected successfully");
                    RemoteId.value = CallObject.remoteId;
                    RemoteLocation.value = CallObject.remoteLocation;
                    RemoteName.value = CallObject.remoteName;
                    scripter.myQueue.objectChangedHandler = null;
                    call.value = 0;
                    CallIDentifier = 0;
                }
            }
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button value="dial" onclick="MakeCall();"></input></td>
            <td>Phonenumber</td>
            <td><input name="Phonenumber" id="Phonenumber"></input></td>
        </tr>
        <tr>
            <td>Local Id</td>
            <td><input name="LocalId" value="" style="width:100"></input></td>
            <td>Local Location</td>
            <td><input name="LocalLocation" value="" style="width:100"></input></td>
            <td>Local Name</td>
            <td><input name="LocalName" value="" style="width:100"></input></td>
        </tr>
        <tr>
            <td>Remote Id</td>
            <td><input name="RemoteId" value="" style="width:100"></input></td>
            <td>Remote Location</td>
            <td><input name="RemoteLocation" value="" style="width:100"></input></td>
            <td>Remote Name</td>
            <td><input name="RemoteName" value="" style="width:100"></input></td>
        </tr>
    </table>
</body>
</html>
```

CallObject.localName Property

Definition

This property returns the name of the logged in user.

Syntax

```
CallObject.localName
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

String

> The name of the logged in user that this call belongs to.

Example

```html
<html>

<head>
    <title>Scripter Object</title>
    <meta name="IS_System_AgentName">

    <script language="JavaScript">
        var CallIDentifier = 0;
        function MakeCall() {
            var CallObject = scripter.createCallObject();
            CallObject.extendedDial(Phonenumber.value, 30, "TRUE", "TRUE");
            CallIDentifier = CallObject.id;
            scripter.myQueue.objectChangedHandler = CheckCallID;
            LocalId.value = CallObject.localId;
            LocalLocation.value = CallObject.localLocation;
            LocalName.value = CallObject.localName;
        }
        function CheckCallID(TypeId, ObjectId) {
            if (ObjectId == CallIDentifier) {
                var CallObject = scripter.callObject;
                CallObject.id = ObjectId;
                if (CallObject.state == 105) {
                    alert("call connected successfully");
                    RemoteId.value = CallObject.remoteId;
                    RemoteLocation.value = CallObject.remoteLocation;
                    RemoteName.value = CallObject.remoteName;
                    scripter.myQueue.objectChangedHandler = null;
                    call.value = 0;
                    CallIDentifier = 0;
                }
            }
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button value="dial" onclick="MakeCall();"></input></td>
            <td>Phonenumber</td>
            <td><input name="Phonenumber" id="Phonenumber"></input></td>
        </tr>
        <tr>
            <td>Local Id</td>
            <td><input name="LocalId" value="" style="width:100"></input></td>
            <td>Local Location</td>
            <td><input name="LocalLocation" value="" style="width:100"></input></td>
            <td>Local Name</td>
            <td><input name="LocalName" value="" style="width:100"></input></td>
        </tr>
        <tr>
            <td>Remote Id</td>
            <td><input name="RemoteId" value="" style="width:100"></input></td>
            <td>Remote Location</td>
            <td><input name="RemoteLocation" value="" style="width:100"></input></td>
            <td>Remote Name</td>
```

```html
            <td><input name="RemoteName" value="" style="width:100"></input></td>
        </tr>
    </table>
</body>
</html>
```

CallObject.remoteId Property

Definition

This property returns the formatted telephone number of the person outside CIC who is making or receiving a call.

Syntax

```
CallObject.remoteId
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

String

The formatted telephone number of the other party.

Example

```html
<html>

<head>
    <title>Scripter Object</title>
    <meta name="IS_System_AgentName">

    <script language="JavaScript">
        var CallIDentifier = 0;
        function MakeCall() {
            var CallObject = scripter.createCallObject();
            CallObject.extendedDial(Phonenumber.value, 30, "TRUE", "TRUE");
            CallIDentifier = CallObject.id;
            scripter.myQueue.objectChangedHandler = CheckCallID;
            LocalId.value = CallObject.localId;
            LocalLocation.value = CallObject.localLocation;
            LocalName.value = CallObject.localName;
        }
        function CheckCallID(TypeId, ObjectId) {

            if (ObjectId == CallIDentifier) {
                var CallObject = scripter.callObject;
                CallObject.id = ObjectId;
                if (CallObject.state == 105) {
                    alert("call connected successfully");
                    RemoteId.value = CallObject.remoteId;
                    RemoteLocation.value = CallObject.remoteLocation;
                    RemoteName.value = CallObject.remoteName;
                    scripter.myQueue.objectChangedHandler = null;
```

```
                    call.value = 0;
                    CallIDentifier = 0;
                }
            }
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button value="dial" onclick="MakeCall();"></input></td>
            <td>Phonenumber</td>
            <td><input name="Phonenumber" id="Phonenumber"></input></td>
        </tr>
        <tr>
            <td>Local Id</td>
            <td><input name="LocalId" value="" style="width:100"></input></td>
            <td>Local Location</td>
            <td><input name="LocalLocation" value="" style="width:100"></input></td>
            <td>Local Name</td>
            <td><input name="LocalName" value="" style="width:100"></input></td>
        </tr>
        <tr>
            <td>Remote Id</td>
            <td><input name="RemoteId" value="" style="width:100"></input></td>
            <td>Remote Location</td>
            <td><input name="RemoteLocation" value="" style="width:100"></input></td>
            <td>Remote Name</td>
            <td><input name="RemoteName" value="" style="width:100"></input></td>
        </tr>
    </table>
</body>
</HTML>
```

CallObject.remoteLocation Property

Definition

This property returns the unformatted telephone number of the person outside CIC who is making or receiving a call.

Syntax

`CallObject.remoteLocation`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

String

> This string contains an unformatted telephone number.

Example

```html
<html>
<head>
    <title>Scripter Object</title>
    <meta name="IS_System_AgentName">

    <script language="JavaScript">
        var CallIDentifier = 0;
        function MakeCall() {
            var CallObject = scripter.createCallObject();
            CallObject.extendedDial(Phonenumber.value, 30, "TRUE", "TRUE");
            CallIDentifier = CallObject.id;
            scripter.myQueue.objectChangedHandler = CheckCallID;
            LocalId.value = CallObject.localId;
            LocalLocation.value = CallObject.localLocation;
            LocalName.value = CallObject.localName;
        }
        function CheckCallID(TypeId, ObjectId) {

            if (ObjectId == CallIDentifier) {
                var CallObject = scripter.callObject;
                CallObject.id = ObjectId;                    if (CallObject.state == 105) {
                    alert("call connected successfully");
                    RemoteId.value = CallObject.remoteId;
                    RemoteLocation.value = CallObject.remoteLocation;
                    RemoteName.value = CallObject.remoteName;
                    scripter.myQueue.objectChangedHandler = null;
                    call.value = 0;
                    CallIDentifier = 0;
                }
            }
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button value="dial" onclick="MakeCall();"></input></td>
            <td>Phonenumber</td>
            <td><input name="Phonenumber" id="Phonenumber"></input></td>
        </tr>
        <tr>
            <td>Local Id</td>
            <td><input name="LocalId" value="" style="width:100"></input></td>
            <td>Local Location</td>
            <td><input name="LocalLocation" value="" style="width:100"></input></td>
            <td>Local Name</td>
            <td><input name="LocalName" value="" style="width:100"></input></td>
        </tr>
        <tr>
            <td>Remote Id</td>
            <td><input name="RemoteId" value="" style="width:100"></input></td>
            <td>Remote Location</td>
            <td><input name="RemoteLocation" value="" style="width:100"></input></td>
            <td>Remote Name</td>
            <td><input name="RemoteName" value="" style="width:100"></input></td>
        </tr>
    </table>
</body></html>
```

CallObject.remoteName Property

Definition

This property returns or sets the name of the caller.

Syntax

```
CallObject.remoteName
```

Usage

Read   Yes

Write   Yes

Value Assigned

String

> A string containing the caller name you wish to assign.

Value Returned

String

> For inbound calls, the name of the person that was looked up in the CIC whitepages. If a name does not exist, the city and state or country of the call will be used if it can be determined.

Example

```html
<html>

    <head>
        <title>Scripter Object</title>
        <meta name="IS_System_AgentName">

        <script language="JavaScript">
            var CallIDentifier = 0;
            function MakeCall() {
                var CallObject = scripter.createCallObject();
                CallObject.extendedDial(Phonenumber.value, 30, "TRUE", "TRUE");
                CallIDentifier = CallObject.id;
                scripter.myQueue.objectChangedHandler = CheckCallID;
                LocalId.value = CallObject.localId;
                LocalLocation.value = CallObject.localLocation;
                LocalName.value = CallObject.localName;
            }
            function CheckCallID(TypeId, ObjectId) {
                if (ObjectId == CallIDentifier) {
                    var CallObject = scripter.callObject;
                    CallObject.id = ObjectId;
                    if (CallObject.state == 105) {
                        alert("call connected successfully");
                        RemoteId.value = CallObject.remoteId;
                        RemoteLocation.value = CallObject.remoteLocation;
                        RemoteName.value = CallObject.remoteName;
                        scripter.myQueue.objectChangedHandler = null;
                        call.value = 0;
                        CallIDentifier = 0;
                    }
                }
            }
        </script>
    </head>

    <body>
        <table>
            <tr>
                <td><input type=button value="dial" onclick="MakeCall();"></input>
                </td>
                <td>Phonenumber</td>
                <td><input name="Phonenumber" id="Phonenumber"></input>
                </td>
```

```
        </tr>
        <tr>
            <td>Local Id</td>
            <td><input name="LocalId" value="" style="width:100"></input></td>
            <td>Local Location</td>
            <td><input name="LocalLocation" value="" style="width:100"></input></td>
            <td>Local Name</td>
            <td><input name="LocalName" value="" style="width:100"></input></td>
        </tr>
        <tr>
            <td>Remote Id</td>
            <td><input name="RemoteId" value="" style="width:100"></input></td>
            <td>Remote Location</td>
            <td><input name="RemoteLocation" value="" style="width:100"></input></td>
            <td>Remote Name</td>
            <td><input name="RemoteName" value="" style="width:100"></input></td>
        </tr>
    </table>
</body>

</html>
```

CallObject.state Property

Definition

This property returns the integer value corresponding to the call state of the call object.

Syntax

`CallObject.state`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

CallState

> The return value represents the Call State string. The table below lists Call State values for recent releases of Customer Interaction Center.

Call State	String Value
Alerting	1
Connected	105
Dialing	103
Disconnected	106
Initializing	100
ManualDialing	102
Offering	101
OnHold	6
Proceeding	104
StationAudio	107

Example

```html
<html>
<head>
    <title>Scripter Object</title>
    <meta name="IS_System_AgentName">

    <script language="JavaScript">
        var CallIDentifier = 0;
        function MakeCall() {
            var CallObject = scripter.createCallObject();
            CallObject.extendedDial(Phonenumber.value, 30, "TRUE", "TRUE");
            CallIDentifier = CallObject.id;
            scripter.myQueue.objectChangedHandler = CheckCallID;
            LocalId.value = CallObject.localId;
            LocalLocation.value = CallObject.localLocation;
            LocalName.value = CallObject.localName;
        }
        function CheckCallID(TypeId, ObjectId) {
            if (ObjectId == CallIDentifier) {
                var CallObject = scripter.callObject;
                CallObject.id = ObjectId;
                if (CallObject.state == 105) {
                    alert("call connected successfully");
                    RemoteId.value = CallObject.remoteId;
                    RemoteLocation.value = CallObject.remoteLocation;
                    RemoteName.value = CallObject.remoteName;
                    scripter.myQueue.objectChangedHandler = null;
                    call.value = 0;
                    CallIDentifier = 0;
                }
            }
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td><input type=button value="dial" onclick="MakeCall();"></input></td>
            <td>Phonenumber</td>
            <td><input name="Phonenumber" id="Phonenumber"></input></td>
        </tr>
        <tr>
            <td>Local Id</td>
```

```html
        <td><input name="LocalId" value="" style="width:100"></input></td>
        <td>Local Location</td>
        <td><input name="LocalLocation" value="" style="width:100"></input></td>
        <td>Local Name</td>
        <td><input name="LocalName" value="" style="width:100"></input></td>
    </tr>
    <tr>
        <td>Remote Id</td>
        <td><input name="RemoteId" value="" style="width:100"></input></td>
        <td>Remote Location</td>
        <td><input name="RemoteLocation" value="" style="width:100"></input></td>
        <td>Remote Name</td>
        <td><input name="RemoteName" value="" style="width:100"></input></td>
    </tr>
  </table>
</body></html>
```

CallObject.stateString Property

Definition

This property retrieves or sets the string value that is displayed as call state information in the CIC client. This can be, but does not have to be, the actual call state. For example, although a call in voice mail has a state of 'Connected', the CallStateString displays 'VoiceMail' for a CIC client user.

Syntax

`CallObject.stateString`

Usage

Read   Yes

Write   Yes

Value Assigned

String

> Any string of characters that you wish to assign to the call's state string attribute.

Value Returned

String

> The current call state string is returned. See <u>callObject.stateChangeHandler</u> for additional details.

CallObject.pauseSecureRecord Method

Definition

This method can be used to avoid recording sensitive information, such as a Social Security number or credit card number, when you are connected to a call interaction. This procedure assumes you are currently connected to a call. You do not have to be currently recording the call interaction. Once pressed, recording will remain paused until CallObject.resumeSecureRecord is invoked.

The Secure Recording Pause Interactions Security right enables you to secure pause a recording of a call. Error handling should be added to the call object to catch any permission issues or any other problems that may prevent the recording from being paused.

Syntax

```
CallObject.pauseSecureRecord();
```

Prototype

```
CallObject.pauseSecureRecord()
```

Input Parameters

None.

Return Values

None.

Example

```
var callObj;
function IS_Event_NewPredictiveCall(CallId) {
    AssignCallIdToObject(CallId);
}
function AssignCallIdToObject(callId) {
    callObj = scripter.createCallObject();
    callObj.id = callId;
    callObj.errorHandler = ErrorHandler;
}function ErrorHandler(ErrorId, ErrorText) {
    alert("Error occured.\n\nError Id: " + ErrorId + "\nError Text: " + ErrorText);
}
function pauseSecureRecord() {
    callObj.pauseSecureRecord();
}
function resumeSecureRecord() {
    callObj.resumeSecureRecord();
}
```

CallObject.resumeSecureRecord Method

Definition

This method resumes recording that was paused by invoking the CallObject.pauseSecureRecord method.

Syntax

```
CallObject.resumeSecureRecord();
```

Prototype

```
CallObject.resumeSecureRecord()
```

Input Parameters

None.

Return Values

None.

Example

```javascript
var callObj;
function IS_Event_NewPredictiveCall(CallId) {
    AssignCallIdToObject(CallId);
}
function AssignCallIdToObject(callId) {
    callObj = scripter.createCallObject();
    callObj.id = callId;
    callObj.errorHandler = ErrorHandler;
}
function ErrorHandler(ErrorId, ErrorText) {
    alert("Error occured.\n\nError Id: " + ErrorId + "\nError Text: " + ErrorText);
}
function pauseSecureRecord() {
    callObj.pauseSecureRecord();
}
function resumeSecureRecord() {
    callObj.resumeSecureRecord();
}
```

campaign object

The **campaign** object encapsulates the Name, ID and Status of a Dialer campaign.

Methods

None.

Callbacks

None.

Properties

campaignName	Returns the display name of the campaign.
campaignId	Returns a GUID that identifies the campaign.
campaignState	Returns an interger value corresponding to the state of the campaign.
campaignStateString	Returns the string value of the current state of the campaign.

campaign.campaignName Property

Definition

This property returns the display name of the campaign.

Syntax

```
campaign.campaignName
```

Usage

Read   Yes
Write   No

Value Assigned

None.

Value Returned

String

   The name of the campaign.

campaign.campaignId Property

Definition

This property returns a GUID that identifies the campaign.

Syntax

```
campaign.campaignId
```

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

String

>   The GUID of the campaign.

campaign.campaignState Property

Definition

This property returns an integer value corresponding to the state of the campaign.

Syntax

campaign.campaignState

Usage

Read   Yes
Write   No

Value Assigned

None.

Value Returned

Integer

> An integer value representing the state of the campaign.

Value	Meaning
-1	invalid
0	Paused
1	Manual Off
2	Manual On
3	Manual On (Schedule only)
4	Auto Off
5	Auto On
6	Auto Off (Schedule only)
7	Auto On (Schedule only)

campaign.campaignStateString Property

Definition

This property returns the string value of the current state of the campaign.

Syntax

`campaign.campaignStateString`

Usage

Read   Yes

Write   No

Value Assigned

None.

Value Returned

String

The string value of the current state of the campaign (Paused/Unpaused, Scheduled calls only, Auto On, Manual On).

ChatObject

ChatObject manipulates chat objects. Chat objects are very similar to call objects. The source examples for call objects can be adapted for chat objects. The methods and properties supported by scripter.chatObject are listed below. Click on a link for information concerning input parameters and return values.

Methods

ChatObject.disconnect	Disconnects the current chat session.
ChatObject.getAttribute	Retrieves the value of the specified chat object attribute.
ChatObject.listen	Allows a CIC user to listen in on a chat.
ChatObject.pause	Pauses recording of the current chat session.
ChatObject.pickup	Picks up (answers) a chat.
ChatObject.private	Makes a chat private so it cannot be seen to or recorded by another CIC user.
ChatObject.record	Records a chat.
ChatObject.sendChatMessage	Sends a chat message to the remote party on behalf of the current user.
ChatObject.setAttribute	Sets the value of specified chat object attribute.

Callbacks

ChatObject.errorHandler	Name of the function to be called when an error occurs. Compatible with scripts for Scripter .NET Client or Interaction Connect.
ChatObject.ObjectDestroyedHandler	ChatObject.objectDestroyedHandler is invoked when the chat object is no longer valid. Compatible with scripts for Scripter .NET Client or Interaction Connect.
ChatObject.ObjectSpecificChangeHandler	Invoked when something changes about this chat object that is not generic to all chat objects. Compatible with scripts for Scripter .NET Client or Interaction Connect.
ChatObject.ObjectSpecificErrorHandler	Invoked when an error occurs with the chat object. Compatible with scripts for Scripter .NET Client or Interaction Connect.
ChatObject.ReceivedFileHandler	This handler is called when a chat object receives a new file. Compatible with scripts for Scripter .NET Client or Interaction Connect.
ChatObject.ReceivedURLHandler	This handler is called when a chat object receives a new URL. Compatible with scripts for Scripter .NET Client or Interaction Connect.
ChatObject.stateChangeHandler	This handler is invoked whenever the call state changes. Compatible with scripts for Scripter .NET Client or Interaction Connect.
ChatObject.SubObjectChangeHandler	This method is called when a sub-object is affected by an event. Compatible with scripts for Scripter .NET Client or Interaction Connect.
ChatObject.requestedAttributeReturnHandler	Allows a script to asynchronously get a chat attribute by first setting this callback and then calling the chatObject.getAttribute method. Use this callback only in scripts for Interaction Connect.
ChatObject.chatObjectInitializedHandler	Allows a script to start a chat after waiting asynchronously for a ChatObject to be created. Use this callback only in scripts for Interaction Connect.

Properties

ChatObject.creationTime	Returns the date and time that the chat object was created.
ChatObject.direction	Returns the direction of the chat (e.g. inbound, outbound, etc.).
ChatObject.id	Assigns or returns the ID of the chat object.
ChatObject.isMonitored	Indicates whether or not a chat is being listened to.
ChatObject.isPaused	Indicates whether recording of a chat has been paused.
ChatObject.isPrivate	Indicates whether is in a private state, meaning that no one can monitor the chat.
ChatObject.isRecording	Indicates whether a chat is being recorded.
ChatObject.lastError	Returns the text of the last error that occurred. This property is automatically cleared before each method or property call on this object.
ChatObject.lastErrorId	Returns the id of the last error that occurred. This property is automatically cleared before each method or property call on this object.
ChatObject.localId	Returns the Station Id of the CIC user associated with the chat.
ChatObject.localLocation	Returns the phone extension of the CIC station participating in a chat.
ChatObject.localName	Returns the name of the connected party in a chat object.
ChatObject.remoteId	Returns the registered name of the remote chat user.
ChatObject.messages	Returns an array of messages about a chat interaction after the chat object has been initialized.
ChatObject.RemoteLocation	Returns the IP address of the chat user.
ChatObject.RemoteName	Retrieves or sets the name of the chat user.
ChatObject.state	Returns the numeric call state value for a chat object. This value indicates the current condition of the chat object.
ChatObject.stateString	Returns or assigns the string value displayed in the State field of a queue (such as the "My Calls" queue). State strings describe the current condition of a chat object.

ChatObject.chatObjectInitializedHandler Callback

Definition

This callback allows a script to start a chat after waiting asynchronously for a ChatObject to be created.

Compatibility

Use this callback only in scripts for Interaction Connect.

Example

```
function main()
{
  var chatObject = scripter.createChatObject();
  chatObject.chatObjectInitializedHandler = uponChatObjectInitialization;
  chatObject.id = exampleId;
}
function uponChatObjectInitialization(chatObject) {
  console.log('printing the chat direction ' + chatObject.direction);}}
```

ChatObject.disconnect Method

Definition

This method disconnects the current chat.

Syntax

```
ChatObject.disconnect();
```

Prototype

```
ChatObject.disconnect()
```

Input Parameters

None.

Return Values

None.

ChatObject.getAttribute Method

Definition

This method retrieves the value of the specified chat object attribute.

Syntax

```
ChatObject.getAttribute(Name);
```

Prototype

```
ChatObject.getAttribute(
  [in] string Name
  [out] string Value
)
```

Input Parameters

Name

> The name of a chat object attribute (e.g., ChatID, StationName, Language) or a custom chat attribute.

Return Values

Value

> The value of the requested attribute is returned as a string value (e.g., 1078924, KevinKPC, Spanish).

ChatObject.listen Method

Definition

This method allows a CIC user to listen in on a chat session.

Syntax

```
ChatObject.listen();
```

Prototype

```
ChatObject.listen()
```

Input Parameters

None.

Return Values

None.

ChatObject.pause Method

Definition

This method pauses recording of the current chat. This method performs a toggling action. To resume recording, call the method again. To stop recording, use ChatObject.record.

Syntax

```
ChatObject.pause();
```

Prototype

```
ChatObject.pause()
```

Input Parameters

None.

Return Values

None.

ChatObject.pickup Method

Definition

This method picks up (answers) a chat.

Syntax

```
ChatObject.pickup();
```

Prototype

```
ChatObject.pickup()
```

Input Parameters

```
None.
```

Return Values

```
None.
```

ChatObject.private Method

Definition

Call this method to make a chat session private so it cannot be listened to or recorded by another CIC user. This method toggles privacy mode on or off. Call the method a second time to toggle privacy mode off.

Syntax

```
ChatObject.private();
```

Prototype

```
ChatObject.private()
```

Input Parameters

None.

Return Values

None.

ChatObject.record Method

Definition

This method records a chat session. To stop recording, call the method a second time. To pause recording, use ChatObject.pause.

Syntax

```
ChatObject.record();
```

Prototype

```
ChatObject.record()
```

Input Parameters

None.

Return Values

None.

ChatObject.sendChatMessage Method

Definition

Sends a chat message to the remote party on behalf of the current user.

Syntax

```
ChatObject.sendChatMessage("Hello, World");
```

Input Parameters

form

> The input parameter is a string containing the text of the chat message.

Example

```
function sendMessage(form)
{ chatObject.sendChatMessage(form.userMsg.value); }
```

ChatObject.setAttribute Method

Definition

This method sets the value of the specified chat object attribute.

Syntax

```
ChatObject.setAttribute(Name, Value);
```

Prototype

```
ChatObject.setAttribute(

  [in] string Name,

  [in] string Value

)
```

Input Parameters

Name

> The name of a chat object attribute (e.g., ChatID, StationName, Language), or the name of a custom (user-defined) call attribute.

Value

> The value that will be assigned to the specified chat object attribute (e.g., 1078924, KevinKPC, Spanish).

Return Values

None.

ChatObject.errorHandler Callback Property

Definition

ChatObject.errorHandler is invoked by when an internal error occurs in the chat object. If you pass the name of a user-defined function to ChatObject.errorHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ChatObject.errorHandler
```

Usage

Read Yes

Write Yes

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, define the errorId, and errorText as arguments to the function. e.g.: function foo(errorId, errorText).

ChatObject.ObjectDestroyedHandler Callback Property

Definition

ChatObject.objectDestroyedHandler is invoked when the chat object is no longer valid. You will not be able to access anything about the object after this function is called on the object. If you pass the name of a user-defined function to ConferenceObject.callobjectDestroyedHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ChatObject.ObjectDestroyedHandler();
```

Usage

Read Yes

Write Yes

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, no special arguments are required. e.g.: function foo().

ChatObject.ObjectSpecificChangeHandler Callback Property

Definition

ChatObject.objectSpecificChangeHandler is invoked when something changes about this chat object that is not generic to all chat objects. You need to query the object itself to see what changed. If you pass the name of a user-defined function to ChatObject.objectSpecificChangeHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ChatObject.objectSpecificChangeHandler();
```

Usage

Read Yes

Write Yes

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, no special arguments are required. e.g.: function foo().

ChatObject.ObjectSpecificErrorHandler Callback Property

Definition

ChatObject.ObjectSpecificErrorHandler is invoked when an error occurs with the chat object. If you pass the name of a user-defined function, that function will be called when an error occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ChatObject.ObjectSpecificErrorHandler();
```

Value Assigned

Function Pointer

> Set this property to a function you wish to be called when an error occurs with the chat object.

Example

```javascript
<script language ="javascript">
    window.onload = Init;
function Init() {
    scripter.chatObject.ObjectSpecificErrorHandler = myFunction;
}
function myFunction() {
    alert("A chat object error occurred.");
} </script>
```

ChatObject.ReceivedFileHandler Callback Property

Definition

This handler is called when a chat object receives a new file.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ChatObject.ReceivedFileHandler (ChatId, User, Filename, FileData);
```

Input Parameters

ChatId

      ID of the chat object.

User

      The user name of the person who sent the file.

ChatId

      ID of the chat object.

Filename

      The fully qualified name of the file received.

FileData

      The data contained in the file.

ChatObject.ReceivedURLHandler Callback Property

Definition

This handler is called when a chat object receives a new URL.

Syntax

```
ChatObject.ReceivedURLHandler();
```

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ChatObject.ReceivedFileHandler (ChatId, User, URL);
```

Input Parameters

ChatId

>ID of the chat object.

User

>The user ID of the user who pushed the URL to the chat.

URL

>A universal resource locator.

ChatObject.stateChangeHandler Callback Property

Definition

This method is called whenever this object's state changes. If you pass the name of a user-defined function to ChatObject.stateChangeHandler, the function will be called whenever the Call State changes.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ChatObject.stateChangeHandler(StateId, StateString)
```

Input Parameters

StateId

>StateID is a number that represents the new call state of the object being watched.

Alerting	1
Connected	105
Dialing	103
Disconnected	106
Initializing	100
ManualDialing	102
Offering	101
OnHold	6
Proceeding	104
StationAudio	107

StateString

StateString is a string that describes the call state.

Initializing	CIC is formatting the telephone number and looking for a line on which to place the outbound call. This state applies to inbound and outbound calls.
Offering	The call has been placed in a queue, but the call is not alerting. CIC is determining if the called party is available to take the call. This state applies to inbound calls only.
Dialing	CIC is dialing the remote telephone number. This state applies to outbound calls only.
Proceeding	The call is proceeding through the outside telephone network. 'Proceeding' is used if a CIC client user has enabled Call Analysis. This state applies to outbound calls only.
Connected	Both parties are connected and are able to speak with each other. This state applies to inbound and outbound calls.
Connected	Is the same as 'Proceeding'.
On Hold	The call is on hold. This state applies to inbound and outbound calls.
Disconnected	The call is no longer active. This state applies to inbound and outbound calls.
Manual Dialing	A telephone handset has been picked up and a dial tone is being generated. This state applies to outbound calls.
Station Audio	An audio clip is being played to one or more CIC client users.
Alerting	A CIC client user is being notified that he or she has an incoming call. This state applies to inbound calls.
Voice Mail	The caller is leaving a voice mail message.

Variables

Function Pointer

A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, you should declare StateID and StateString as arguments. E.g. function foo(StateID, StateString).

ChatObject.SubObjectChangeHandler Callback Property

Definition

Read   Yes

Write   Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

`ChatObject.subObjectChangeHandler(TypeId, CallID, ChangeId)`

Input Parameters

TypeId

TypeId is an integer that represents the type of sub-object.

2	Call Object
19	Chat Object
20	Conference Object
70	Generic Object

CallID

The Id of the sub-object.

ChangeId

An integer that represents the type of change that occurred in a conference object.

80	A party call changed. This indicates that something changed about the call. For example, placing a call on hold changes the call.
82	A party was deallocated (destroyed). Two minutes have passed since the call was disconnected, and it is being removed from the conference.
83	The call was just added to the conference.
84	The call was just removed from the conference (e.g. transferred to another queue).

Value Assigned

Function Pointer

A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, you may define the following arguments in the function. e.g.: function foo(Type ID, CallID, ChangeID).

Example

The [messages](#) property of the chatObject returns an array of messages of that chatInteraction after the chatObject had been

initialized. The SubObjectChangeHandler can be an assigned callback that gets alerted with new messages when the chat Interaction receives new messages.

```
function chatInitialized(){
if(chatObject.messages !== undefined)
{ renderChatBox(chatObject.messages); chatObject.SubObjectChangeHandler = messagesChanged; }
}
function messagesChanged()
{ renderChatBox(chatObject.messages); }
```

Each message is an object with the following properties:

chatMember.displayName

chatMember.chatMemberType

chatMember.interactionId

chatMember.userId

messageType

text

timestamp

ChatObject.creationTime Property

Definition

This property returns the date and time that the chat object was created.

Syntax

ChatObject.creationTime

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Date

The date and time are passed as a script Date object.

ChatObject.direction Property

Definition

This property allows you to query the direction of the chat (e.g. inbound, outbound, etc.).

Syntax

`ChatObject.direction`

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Direction

An integer value representing the chat direction.

0	None
1	Inbound Call
2	Outbound Call

ChatObject.id Property

Definition

The ID property assigns or returns the ID of the chat object.

Syntax

`ChatObject.Id`

Usage

Read Yes

Write Yes

Value Assigned

pVal

A chat object ID, passed as string data.

Value Returned

String

The chat ID is returned as a number.

ChatObject.isMonitored Property

Definition

This property indicates whether or not a chat session is being listened to.

Syntax

`ChatObject.isMonitored`

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

Returns True if the chat is being listened to. Otherwise, returns False.

ChatObject.isPaused Property

Definition

This property indicates whether recording of this chat session has been paused.

Syntax

```
ChatObject.isPaused
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

> If recording has been paused, returns True. Otherwise, returns False.

ChatObject.isPrivate Property

Definition

This property indicates whether the chat is in a private state, meaning that no one can listen in on (monitor) the session.

Syntax

```
ChatObject.isPrivate
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

> The property returns True if the chat is in a private state; otherwise, returns False.

ChatObject.isRecording Property

Definition

This property indicates whether the chat session is being recorded.

Syntax

```
ChatObject.isRecording
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

    Returns True if the chat session is being recorded; otherwise, returns False.

ChatObject.lastError Property

Definition

This property retrieves the text of the last error that occurred in the ChatObject. Each time a method or property is called on the ChatObject, this value is cleared.

Syntax

```
ChatObject.lastError
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

String

    The text message of the last error that occurred.

ChatObject.lastErrorId Property

Definition

This property retrieves the id of the last error that occurred in the ChatObject. Each time a method or property is called on the ChatObject, this value is cleared.

Syntax

```
ChatObject.lastErrorId
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Integer

> The id of the last error that occurred.

ChatObject.localId Property

Definition

This property returns the Station Id of the CIC user associated with the chat.

Syntax

```
CallObject.localId
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

localID

> When you retrieve this value, a string identifying the Station Id is returned.

ChatObject.localLocation Property

Definition

This property returns the telephone extension of the CIC station participating in a chat.

Syntax

`ChatObject.LocalLocation`

Usage

Read Yes

Write No

Value Returned

String

> The telephone extension of the CIC station participating in a chat is returned as a string.

ChatObject.localName Property

Definition

The LocalName property returns the name of the connected party in a chat object.

Syntax

`ChatObject.LocalName`

Usage

Read Yes

Write No

Value Returned

String

> The name of the chat user.

ChatObject.messages Property

Definition

The messages property of the chatObject returns an array of messages of that chat interaction after the chatObject had been initialized. The [SubObjectChangeHandler](#) can be an assigned callback that gets alerted with new messages when the chat Interaction receives new messages.

Syntax

```
function chatInitialized(){
if(chatObject.messages !== undefined)
{ renderChatBox(chatObject.messages); chatObject.SubObjectChangeHandler = messagesChanged; }
}
function messagesChanged()
{ renderChatBox(chatObject.messages); }
```

Each message is an object with the following properties:

```
chatMember.displayName
```

```
chatMember.chatMemberType
```

```
chatMember.interactionId
```

```
chatMember.userId
```

```
messageType
```

```
text
```

```
timestamp
```

ChatObject.remoteId Property

Definition

This property returns the registered name of the remote chat user.

Syntax

```
ChatObject.remoteId
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

String

> A string value representing the registered name of the remote chat user.

ChatObject.requestedAttributeReturnHandler Callback

Definition

This callback allows a script to asynchronously get a chat attribute by first setting requestedAttributeReturnHandler and then calling chatObject.getAttribute. The chatObject will trigger the requestedAttributeReturnHandler handler after it has pulled down the necessary interaction attribute information from the server.

Compatibility

Use this callback only in scripts for Interaction Connect.

Syntax

```
ChatObject.requestedAttributeReturnHandler(attributeName, attributeValue);
```

Input Parameters

Name

The name of a chat object attribute (e.g., ChatID, StationName, Language) or a custom chat attribute.

Return Values

Value

The value of the requested attribute is returned as a string.

Example

```
function main()
{
  var chatObject = scripter.createChatObject();
  chatObject.chatObjectInitializedHandler = uponChatObjectInitialization;
  chatObject.id = exampleId;
}
function uponChatObjectInitialization(chatObject)
{
  console.log('printing the chat direction ' + chatObject.direction);
  chatObject.requestedAttributeReturnHandler = uponAttributeReception;
  chatObject.getAttribute('IS_ATTR_EXAMPLE');
}
function uponAttributeReception(attributeName, attributeValue)
{
  console.log('The value of ' + attributeName + ' is ' + attributeValue);
}
```

ChatObject.remoteLocation Property

Definition

The RemoteLocation property returns the IP address of the chat user.

Syntax

```
ChatObject.remoteLocation
```

Usage

Read Yes

Write No

Value Returned

String

> A string value representing the IP address of the remote chat user.

ChatObject.remoteName Property

Definition

The ChatObject.remoteName property allows you to retrieve the user name entered by the person who requested the chat. You can also assign a name.

Syntax

```
ChatObject.remoteName
```

Usage

Read Yes

Write Yes

Value Assigned

pVal

> Any string value that you wish to assign.

Value Returned

String

> The name of the person who requested the chat.

ChatObject.state Property

Definition

ChatObject.state returns the numeric call state> value for a chat object. This value indicates the current condition of the chat object. To return a string value that describes the state, use the ChatObject.StateString property.

Syntax

```
ChatObject.state
```

Usage

Read Yes

Write No

Value Returned

Integer

The value returned is the call state value for the chat object. Possible values are shown below.

1	ALERTING
2	CONNECTED
3	CLIENT_CONNECT
4	HELD
5	INTERNAL_DISCONNECT
6	EXTERNAL_DISCONNECT

ChatObject.stateString Property

Definition

The StateString property returns or assigns the string value displayed in the State field of a queue (such as the "My Calls" queue). State strings describes the current condition of a chat object. To return a numeric value that describes the state, use the ChatObject.state property.

Syntax

`ChatObject.stateString`

Usage

Read Yes

Write Yes

Value Assigned

String

> This string value describes the current condition of the chat. Possible values are:

Alerting
Voicemail
Connected
Disconnected
Initializing
Proceeding
ACD - Connected (Agent XX)
Any User-defined string

Value Returned

String

> When the StateString property is read, a string containing the call state is returned.

ConferenceObject

The ConferenceObject class manages conference calls. It allows you to begin a conference call, add calls to the conference, and disconnect parties from the conference. Properties of the ConferenceObject provide notification, handling, and the ability to enumerate objects in the conference.

Methods

ConferenceObject.add	This method adds a call into a conference.
ConferenceObject.create	Used to begin a conference call.
ConferenceObject.disconnectParty	This method disconnects a party from a conference.

Callbacks

ConferenceObject.errorHandler	Name of the function to be called when an error occurs.
ConferenceObject.objectDestroyedHandler	Name of the function to be called when the object is no longer valid. You will not be able to access anything about the object after this function is called.
ConferenceObject.objectSpecificChangeHandler	Name of the function to be called when something changes about an object that is not generic to all conference objects. You need to query the object itself to see what changed.
ConferenceObject.stateChangeHandler	Name of the function to be called when the call state of object being watched changes.
ConferenceObject.subObjectChangeHandler	Name of the function to be called when a subobject changes. This method is used to monitor queue objects that are made up of other objects. For example, Conference objects consist of multiple call objects.
ConferenceObject.conferenceObjectInitializedHandler	This callback is invoked when the conference object has initialized. It is compatible with scripts for Interaction Connect only.
ConferenceObject.conferenceStartedHandler	This callback is invoked when the conference call has started. It is compatible with scripts for Interaction Connect only.

Properties

ConferenceObject.id	The ID property assigns or returns the ID of the conference object.
ConferenceObject.lastError	Returns the text of the last error that occurred. This property is automatically cleared before each method or property call on this object.
ConferenceObject.lastErrorId	Returns the id of the last error that occurred. This property is automatically cleared before each method or property call on this object.

Enumerations

ConferenceObject.startMemberIdsEnum	Returns an enumeration of object ids for each object in the conference.

ConferenceObject.add Method

Definition

This method adds a call into a conference.

Syntax

```
ConferenceObject.add(CallObject);
```

Prototype

```
ConferenceObject.add(
    [in] CallObject CallObject
)
```

Input Parameters

CallObject

>   The call object to add to the conference.

Return Values

None.

Example

```javascript
function CreateConference() {
    scripter.callObject.dial("5007", false);
    alert("Click OK to dial second number");
    scripter.conferenceObject.create(scripter.callobject);
    var objCall2 = scripter.createCallObject();
    objCall2.dial("555-1212", false);
    alert("Click OK to join conference.");
    scripter.conferenceObject.add(objCall2);
}
```

ConferenceObject.create Method

Definition

Creates a conference call. This method has 1 or 2 input parameters, depending upon whether the script will run in Scripter .NET or Interaction Connect. To create a conference in an Interaction Connect script, two interaction id's must be passed to this method. Scripter .NET requires a single interaction ID to be passed.

Creating a conference in Interaction Connect scripts

Syntax

```
conferenceObject.create(callid1, callid2);
```

Input Parameters

callid1

Interaction ID of the first party in the conference call.

Interaction ID of the first party in the conference call.

callid2

Interaction ID of the second party in the conference call.

Example

This example assumes the agent is currently on a dialer interaction, and would like to initiate a conference with their Accounting department by selecting the "Conference Accounting" button.

```
<input type="button" onclick="conferenceAccounting()">Conference Accounting</input>
<script>
var callObj1 = scripter.createCallObject();
var callObj2 = scripter.createCallObject();
var confObj = scripter.createConferenceObject();
function conferenceAccounting() {
    // Assign the current dialer interaction to callObj1
    callObj1.id = IS_Attr_CallID.value;
    // Specify a callback to be triggered once the call has successfully been placed to the Accounting
department.
    callObj2.callObjectInitializedHandler = startConference;
    // Place the call to Accounting
    callObj2.dial("1234");
}
function startConference() {
    // Start the conference between the customer and accounting by providing the interaction id for each of
the calls.
    confObj.create(callObj1.id, callObj2.id);
}
</script>
```

Creating a conference in Scripter .NET scripts

Syntax

```
ConferenceObject.create(CallObject);
```

Prototype

```
ConferenceObject.create(
  [in] CallObject CallObject
)
```

Input Parameters

CallObject

The call object to create the conference around. If the call object's id is −1, then an empty conference will be created.

Return Values

None.

Example

```
function CreateConference()
{
  scripter.callObject.dial("5007", false);
  alert("Click OK to dial second number");
  scripter.conferenceObject.create(scripter.callobject);
  var objCall2 = scripter.createCallObject();
  objCall2.dial("555-1212", false);
```

```
   alert("Click OK to join conference.");
   scripter.conferenceObject.add(objCall2);
}
```

ConferenceObject.disconnectParty Method

Definition

This method disconnects a party from a conference.

Syntax

```
ConferenceObject.disconnectParty(CallID);
```

Prototype

```
ConferenceObject.disconnectParty(
   [in] String CallID
)
```

Input Parameters

CallID

The call id of the call to disconnect from the conference.

Return Values

None.

ConferenceObject.errorHandler Callback Property

Definition

ConferenceObject.errorHandler is invoked by when an internal error occurs in the conference object. If you pass the name of a user-defined function to ConferenceObject.errorHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ConferenceObject.errorHandler();
```

Usage

Read Yes

Write Yes

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, define the errorId, and errorText as arguments to the function. e.g.: function foo(errorId, errorText).

ConferenceObject.objectDestroyedHandler Callback Property

Definition

ConferenceObject.objectDestroyedHandler is invoked when the conference object is no longer valid. You will not be able to access anything about the object after this function is called on the object. If you pass the name of a user-defined function to ConferenceObject.callobjectDestroyedHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ConferenceObject.objectDestroyedHandler();
```

Usage

Read Yes

Write Yes

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user -defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, no special arguments are required. e.g.: function foo().

ConferenceObject.objectSpecificChangeHandler Callback Property

Definition

ConferenceObject.objectSpecificChangeHandler is invoked when something changes about this conference object that is not generic to all conference objects. You need to query the object itself to see what changed. If you pass the name of a user-defined function to ConferenceObject.objectSpecificChangeHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ConferenceObject.objectSpecificChangeHandler();
```

Usage

Read Yes

Write Yes

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, no special arguments are required. e.g.: function foo().

ConferenceObject.stateChangeHandler Callback Property

Definition

This method is called whenever this object's state changes . If you pass the name of a user-defined function to ConferenceObject.stateChangeHandler, the function will be called whenever the conference state changes.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
ConferenceObject.stateChangeHandler(StateId, StateString);
```

StateId

> StateID is a number that represents the new call state of the object being watched.

Alerting	1
Connected	105
Dialing	103
Disconnected	106
Initializing	100
ManualDialing	102
Offering	101
OnHold	6
Proceeding	104
StationAudio	107

StateString

StateString is a string that describes the call state.

Initializing	CIC is formatting the telephone number and looking for a line on which to place the outbound call. This state applies to inbound and outbound calls.
Offering	The call has been placed in a queue, but the call is not alerting. CIC is determining if the called party is available to take the call. This state applies to inbound calls only.
Dialing	CIC is dialing the remote telephone number. This state applies to outbound calls only.
Proceeding	The call is proceeding through the outside telephone network. 'Proceeding' is used if a CIC client user has enabled Call Analysis. This state applies to outbound calls only.
Connected	Both parties are connected and are able to speak with each other. This state applies to inbound and outbound calls. Connected is the same as Proceeding.
On Hold	The call is on hold. This state applies to inbound and outbound calls.
Disconnected	The call is no longer active. This state applies to inbound and outbound calls.
Manual Dialing	A telephone handset has been picked up and a dial tone is being generated. This state applies to outbound calls.
Station Audio	An audio clip is being played to one or more CIC client users.
Alerting	A CIC client user is being notified that he or she has an incoming call. This state applies to inbound calls.
Voice Mail	The caller is leaving a voice mail message.

Variables

Function Pointer

A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user -defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, you should declare StateID and StateString as arguments. e.g.: function foo(StateID, StateString).

ConferenceObject.subObjectChangeHandler Callback Property

Definition

This method is called when a sub-object (such as a call in a conference) is affected by an event . If you pass the name of a user-defined function to ConferenceObject.stateChangeHandler, the function will be called whenever the call state changes. This method is used to monitor queue objects that are made up of other objects. For example, a conference object consists of multiple call objects.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

ConferenceObject.subObjectChangeHandler(ChangedAttributes, MemberInteraction)

Parameters

ChangedAttributes

An object containing the specific attributes of the conference member that changed.

MemberInteraction

The interaction object of the conference member that changed.

Value Assigned

Function Pointer

A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, you may define the following arguments in the function. e.g.: function foo(Type ID, CallID, ChangeID).

Example

The following example assumes a conference has already been started. In the example, the subObjectCahngedHandler callback assigned to the conference object will log to the browser the Interaction Id and state of the member interaction as well as each attribute that has changed.

```
var conferenceMemberChanged = function(changedAttributes, memberInteraction) {
    console.log("Member Interaction Id: ", memberInteraction.interactionId, "Member state: ",
memberInteraction.stateDisplayString);
    changedAttributes.forEach(function(attribute) {
        console.log("Attribute Name: ", attribute.attributeName, "Previous Value: ", attribute.oldValue, "New
Value: ", attribute.newValue);
    }
    }
    scripter.conferenceObject.subObjectChangedHandler = conferenceMemberChanged;
```

ConferenceObject.conferenceObjectInitializedHandler Callback Property

Definition

This callback is invoked when the conference object has initialized.

Compatibility

This callback is compatible with scripts for Interaction Connect only.

Syntax

```
ConferenceObject.conferenceObjectInitializedHandler();
```

Example

This example initializes the conference object by assigning the Interaction ID of an existing conference to the newly created conference object. A function is assigned to the conferenceObjectInitializedHandler and gets called once the conference object has initialized.

```
var conferenceInitializedMessage = function() { console.log("The conference object has been initialized.") };
var conference = scripter.createConferenceObject();
conference.conferenceObjectInitializedHandler = conferenceInitializedMessage;
conference.id = "1234567890";
```

ConferenceObject.conferenceStartedHandler Callback Property

Definition

This callback is invoked when the conference call has started.

Compatibility

This callback is compatible with scripts for Interaction Connect only.

Syntax

```
ConferenceObject.conferenceStartedHandler();
```

Example

This example starts a conference with the current dialer call and the user at extension "456". Once the conference has started, the conferenceStartedHandler is invoked.

```
var callObj1 = scripter.createCallObject();
var callObj2 = scripter.createCallObject();
var confObj = scripter.createConferenceObject();
callObject1.id = IS_Attr_CallID.value;
callObject2.dial("456");
var conferenceStarted = function() { console.log("The conference has started") };
confObj.conferenceStartedHandler = conferenceStarted;
confObj.create(callObj1.id, callObj2.id);
```

ConferenceObject.id Property

Definition

The ID property assigns or returns the ID of the conference object.

Syntax

```
ConferenceObject.Id
```

Usage

Read Yes

Write Yes

Value Assigned

pVal

>A conference object ID, passed as string data.

Value Returned

String

>The conference object ID is returned as a string.

ConferenceObject.lastError Property

Definition

This property retrieves the text of the last error that occurred in the ConferenceObject. Each time a method or property is called on the ConferenceObject, this value is cleared.

Syntax

```
ConferenceObject.lastError
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

String

>The text message of the last error that occurred.

ConferenceObject.lastErrorId Property

Definition

This property retrieves the id of the last error that occurred in the ConferenceObject. Each time a method or property is called on the ConferenceObject, this value is cleared.

Syntax

```
ConferenceObject.lastErrorId
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Integer

The id of the last error that occurred.

ConferenceObject.startMemberIdsEnum Enumeration Property

Definition

This property returns an enumeration of object ids for each object in the conference. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Syntax

```
ConferenceObject.startMemberIdsEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Java Enumeration

The enumeration containing the list of ConferenceObjects.

Java Enumeration Example (JavaScript)

```javascript
var members = ConferenceObject.startMemberIdsEnum;
while (members.hasMoreElements()) {
    var memberId = members.nextElement();
    alert(memberId);
}
```

Queue object

The Queue object connects to CIC call queues, and provides access to properties supported by queue objects. An additional queue object, >scripter.myQueue is functionally equivalent to scripter.queue, except that the current user's queue is preattached in the myQueue object.

Methods

Queue.connect	This method connects a queue object to a specific CIC queue.

Callbacks

Queue.callObjectAddedHandler	The name of the function to be invoked when a CallObject is added to the queue. This is different from Queue.objectAddedHandler in that it only responds to CallObjects.
Queue.errorHandler	Name of the function to be called when an error occurs in this object.
Queue.objectAddedHandler	Queue.objectAddedHandler allows you to specify the name of a procedure that will be called when an object is added to a queue.
Queue.objectChangedHandler	Queue.objectChangedHandler allows you to specify the name of a procedure that will be called when a queue object is changed.
Queue.objectRemovedHandler	Queue.objectRemovedHandler allows you to specify the name of a procedure that will be called when an queue object is removed.

Properties

Queue.activeMonitor	This property indicates whether or not someone is actively monitoring this queue (i.e., you should receive ACD calls).
Queue.lastError	This property returns a string describing an error condition affecting the queue object.
Queue.lastErrorId	This property returns the number of an error condition affecting the queue object.
Queue.name	Returns the name of a user, workstation, workgroup, or line queue.
Queue.type	Returns an integer identifying the type of queue (station, user, workstation, or line).

Enumerations

Queue.startCallObjectsEnum	Queue.startCallObjectsEnum returns a new enumeration of all call objects currently within the queue.
Queue.startChatObjectsEnum	Queue.startChatObjectsEnum returns a new enumeration of all chat objects currently within the queue.
Queue.startConferenceObjectsEnum	Queue.startConferenceObjectsEnum returns a new enumeration of all conference objects currently within the queue.
Queue.startObjectIdsEnum	Queue.startObjectIdsEnum returns a new enumeration of all objects ids currently within the queue.

Queue.connect Method

Definition

This method connects to a specific CIC queue. Starting in 2018 R3, this method is asynchronous, meaning that it can accept a single optional callback argument that takes no parameters. In addition, Queue.connect no longer supports Line Queue as a Type parameter. Scripts for Scripter .NET do not need to specify a callback, but scripts for Connect must specify it.

Syntax

```
Queue.connect(Type, Name);
```

Prototype

```
Queue.connect(
  [in] int Type,
  [in] string Name
)
```

Input Parameters

Type

Type is an integer representing a queue type. Valid values for queue types are:

3	Station queue
9	User queue
10	Workgroup queue

Name

The name of the queue that you wish to connect to the queue object.

Return Values

None.

Example

Connect to a queue, using a callback to wait asynchronously for the connection to be made.

```
// Connect a queue
var queue = scripter.myQueue;
queue.connect(9, 'user1', function() {
  queue.startObjectIdsEnum(function(result) {
    while (result.hasMoreElements()) {
      console.log('Id:', result.nextElement());
    }
  });
});
```

See also Queue.callObjectAddedHandler.

Queue.callObjectAddedHandler Callback Property

Definition

Queue.callObjectAddedHandler allows you to define a function that executes when a call object is added to a queue. This method is called only when a call object is added to the monitored queue.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
Queue.callObjectAddedHandler(CallObject)
```

Parameters

CallObject

> The actual call object that was added is passed.

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". Function pointers are used when you wish to change the value of a property. When defining your custom function, you should define CallObject as the argument to the function. e.g.: function foo(CallObject).

Example

```html
<html>
<head>
    <title>Scripter Object</title>
    <meta name="IS_System_AgentName">

    <script language="JavaScript">
        var QueueEx = scripter.createQueue();

        function ConnectQueue() {
            QueueEx.connect(QueueType.value, QueueName.value);
            QueueEx.callObjectAddedHandler = CallObjectAdded;
            QueueEx.objectAddedHandler = QueueObjectAdded;
            QueueEx.objectChangedHandler = QueueObjectChanged;
            QueueEx.objectRemovedHandler = QueueObjectRemoved;
            alert("Connection established to Queue: " + QueueEx.name);
        }
        function CallObjectAdded(callobj) {
            alert("call was added to user queue: " + callobj.id);
        }
        function QueueObjectAdded(TypeId, ObjectId) {
            alert("object was added: " + ObjectId + "\n Type: " + TypeId);
        }
        function QueueObjectRemoved(TypeId, ObjectId) {
            alert("object was removed: " + ObjectId + "\n Type: " + TypeId);
        }
        function QueueObjectChanged(TypeId, ObjectId) {
            alert("object was changed: " + ObjectId + "\n Type: " + TypeId);
        }
```

```html
        </script>
</head>
<body>
    <table>
        <tr>
            <td>
                <input type="button" value="Connect Queue" onclick="ConnectQueue();"></input>
            </td>
            <td>Queue Name</td>
            <td><input name="QueueName" value="" style="width:100"></input>
            </td>
            <td>Queue Type</td>
            <td><input name="QueueType" value="9" style="width:100"></input>
            </td>
        </tr>
    </table>
</body>
</html>
```

Queue.errorHandler Callback Property

Definition

Queue.errorHandler is invoked by when an internal error occurs in the Queue object. If you pass the name of a user-defined function to Queue.errorHandler, the function will be called when this event occurs. In other words, it is called when an error occurs during the processing of a queue operation. The HRESULT and the error text are passed as parameters.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client. Do not use this callback in scripts for Interaction Connect.

Syntax

Queue.errorHandler(errId, ErrText)

Parameters

errId

The HRESULT error number.

ErrText

Textual description of the error.

Value Assigned

Function Pointer

A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user -defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". Function pointers are used when you wish to change the value of a property. When defining your custom function, you may define the following arguments in the function. e.g.: function foo(errID, errText).

Example

```
scripter.myQueue.errorHandler = foo;
function foo(errID, errText); {
    alert("Error " + errID + ":" + errText);
}
```

Queue.objectAddedHandler Callback Property

Definition

Queue.objectAddedHandler allows you to specify the name of a procedure that will be called when an object is added to a queue. See queue.callObjectAddedHandler for example code.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
Queue.objectAddedHandler(TypeId, ObjectId)
```

Parameters

TypeId

An integer representing the type of object that was added to the queue.

2	Call Object
19	Chat Object

ObjectId

The object's identifier.

Value Assigned

Function Pointer

A function pointer is the address in memory where a user->defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". Function pointers are used when you wish to change the value of a property. When defining your custom function, you should define TypeId and ObjectId as arguments to the function. e.g.: function foo(TypeId, ObjectId).

Queue.objectChangedHandler Callback Property

Definition

Queue.objectChangedHandler allows you to specify the name of a procedure that will be called when a queue object changes within the queue.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
Queue.objectChangedHandler(TypeId, ObjectId)
```

Parameters

TypeId

An integer representing the type of object that changed in the queue.

2	Call Object
19	Chat Object

ObjectId

The object's identifier.

Value Assigned

Function Pointer

A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". Function pointers are used when you wish to change the value of a property. When defining your custom function, you should define TypeId and ObjectId as arguments to the function. e.g.: function foo(TypeId, ObjectId).

Example

See queue.callObjectAddedHandler for example code.

Queue.objectRemovedHandler Callback Property

Definition

Queue.objectRemovedHandler allows you to specify the name of a procedure that will be called when an queue object is removed from the queue.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
Queue.objectRemovedHandler(TypeId, ObjectId)
```

Parameters

TypeId

An integer representing the type of object that was removed from the queue.

2	Call Object
19	Chat Object

ObjectId

The object's identifier.

Value Assigned

Function Pointer

A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". Function pointers are used when you wish to change the value of a property.

When defining your custom function, you should define TypeId and ObjectId as arguments to the function. e.g.: function foo(TypeId, ObjectId).

Example

See queue.callObjectAddedHandler.

Queue.activeMonitor Property

Definition

This property sets / indicates whether or not the queue is actively being monitored.

Syntax

`Queue.activeMonitor`

Usage

Read Yes

Write Yes

Value Assigned

Boolean

> Set this property to True if you are monitoring the queue.

Value Returned

Boolean

> Returns True if the queue is actively being monitored; otherwise, returns False.

Queue.lastError Property

Definition

This property retrieves the text of the last error that occurred in the Queue object. Each time a method or property is called on the Queue object, this value is cleared.

Compatibility

This property is compatible with scripts for Scripter .NET Client. Do not use this property in scripts for Interaction Connect.

Syntax

`Queue.lastError`

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Integer

Error message describing the problem with the queue object.

Queue.lastErrorId Property

Definition

This property retrieves the numeric id of the last error that occurred in the Queue object. Each time a method or property is called on the Queue object, this value is cleared.

Compatibility

This property is compatible with scripts for Scripter .NET Client. Do not use this property in scripts for Interaction Connect.

Syntax

```
Queue.lastErrorId
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

String

Numeric error id describing the problem with the queue object.

Queue.name Property

Definition

This property returns the name of a user, workstation, workgroup, or line queue.

Syntax

`Queue.name`

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

String

A string containing the name of a station, workgroup, user, or line queue.

Example

See [queue.callObjectAddedHandler](#) for example code.

Queue.type Property

Definition

This property returns an integer identifying the type of queue (station, user, workstation, or line).

Syntax

```
Queue.type
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Integer

Possible return values:

3	Station queue
9	User queue
10	Workgroup queue
15	Line queue

Queue.startCallObjectsEnum Enumeration Property

Definition

This property returns an enumeration of CallObjects for queue. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Starting with 2018 R3, the Queue.startCallObjectsEnum, Queue.startChatObjectsEnum, Queue.startConferenceObjectsEnum and Queue.startObjectIdsEnum properties now accept an optional callback (for use with Connect scripts only) whose single parameter contains the result.

Syntax

```
Queue.startCallObjectsEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Java Enumeration

> The enumeration containing the list of CallObjects.

Java Enumeration Example (JavaScript)

```javascript
// disconnect call objects in the queue
var calls = scripter.myQueue.startCallObjectsEnum;
while (calls.hasMoreElements()) {
    var CallObject = calls.nextElement();
    CallObject.disconnect();
}
```

Interaction Connect Example

```javascript
function queue.startObjectIdsEnum(function(result) {
  try {
    var objects = [];
    while (result.hasMoreElements()) {
      objects.push('<li>' + result.nextElement() + '</li>');
    }
    var tdHtml = '<ul>' + objects.join('') + '</ul>';    doc.getElementById('queue-ids-td').innerHTML =
tdHtml;
    console.log('callback complete.');
  } catch (error) {
    console.log('error in startObjectIdsEnum:', error);
  }
});
```

Queue.startChatObjectsEnum Enumeration Property

Definition

This property returns an enumeration of ChatObjects for queue. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Starting with 2018 R3, the Queue.startCallObjectsEnum, Queue.startChatObjectsEnum, Queue.startConferenceObjectsEnum and Queue.startObjectIdsEnum properties now accept an optional callback (for use with Connect scripts only) whose single parameter contains the result. See Interaction Connect Example in Queue.startCallObjectsEnum Enumeration Property for an example.

Syntax

```
Queue.startChatObjectsEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Java Enumeration

   The enumeration containing the list of ChatObjects.

Java Enumeration Example (JavaScript)

```javascript
// disconnect call objects in the queue
var calls = scripter.myQueue.startCallObjectsEnum;
while (calls.hasMoreElements()) {
    var CallObject = calls.nextElement();
    CallObject.disconnect();
}
```

Queue.startConferenceObjectsEnum Enumeration Property

Definition

This property returns an enumeration of ConferenceObjects for queue. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean True if this enumeration contains more elements.

Starting with 2018 R3, the Queue.startCallObjectsEnum, Queue.startChatObjectsEnum, Queue.startConferenceObjectsEnum and Queue.startObjectIdsEnum properties now accept an optional callback (for use with Connect scripts only) whose single parameter contains the result. See Interaction Connect Example in Queue.startCallObjectsEnum Enumeration Property for an example.

Syntax

```
Queue.startConferenceObjectsEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Java Enumeration

> The enumeration containing the list of ConferenceObjects.

Java Enumeration Example (JavaScript)

```javascript
// search for conference objects in the queue
var conferences = scripter.myQueue.startConferenceObjectsEnum;
if (conferences.hasMoreElements()) {
    alert("A conference is in the queue.");
}
```

Queue.startObjectIdsEnum Enumeration Property

Definition

This property returns an enumeration of object ids for all objects in the queue. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Starting with 2018 R3, the Queue.startCallObjectsEnum, Queue.startChatObjectsEnum, Queue.startConferenceObjectsEnum and Queue.startObjectIdsEnum properties now accept an optional callback (for use with Connect scripts only) whose single parameter contains the result. See Interaction Connect Example in Queue.startCallObjectsEnum Enumeration Property for an example.

Syntax

Queue.startObjectIdsEnum

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Java Enumeration

The enumeration contains the list of object ids.

Java Enumeration Example (JavaScript)

```
var i = 0;
var ObjectIds = new Array();
var objects = scripter.myQueue.startObjectIdsEnum;
while (objects.hasMoreElements()) {
    ObjectIds[i++] = objects.nextElement();
}
```

User Object

The User object is used to obtain information about a specific CIC user, such as the list of user, station, line, and workgroup queues that the user can view and modify, the user's status, logged in state, etc.

Methods

None

Callbacks

User.errorHandler	Name of the script function to be invoked by when an internal error occurs in the User object.
User.statusChangeHandler	Name of the script function to be called when the agent's status changes.
User.userChangeHandler	Name of the script function to be called when any of the agent's configuration attributes changes, except the agent's status.
User.userLoginChangeHandler	Name of the script function to be called when the agent logs in or out of the system. Login will be true on a login message and false on a logout message.

Properties

User.canListen	Determines if the agent can listen in on calls.
User.canMakePrivate	Determines if the agent's calls can be made private.
User.canRecord	Determines if the agent can record calls.
User.extension	The agent's CIC extension number.
User.id	The CIC user name of this agent.
User.isDND	Determines if the agent's status is in a Do Not Disturb state.
User.isLoggedIn	Determines if the agent is logged in.
User.isOnPhone	Determines if the agent is on the phone.
User.lastError	This property returns a string describing an error condition affecting the User object.
User.lastErrorId	This property returns the number of an error condition affecting the User object.
User.name	The CIC display name of the agent.
User.statusMessage	Sets or returns the agent's status indicator (e.g. Available).
User.untilDateTime	The date and time that the agent will return from an unavailable status condition.

Enumerations

User.startAccessibleQueuesEnum	Returns an enumeration of all accessible queues (e.g., user queues or workgroups with queues) the current agent has rights to view and modify.
User.startAvailableCampaignObjectEnum	Returns an enumeration of campaigns which are available for agent to logon to.
User.startAvailableStatusMessagesEnum	Returns an enumeration of available status messages.
User.startLoggedInStationsEnum	Returns an enumeration of the logged in stations for this agent.
User.startViewableWorkgroupsEnum	Returns an enumeration of all workgroups the agent has rights to view.
User.startWorkgroupsEnum	Creates a list of workgroups the current agent is a member of.

User.errorHandler Callback Property

Definition

User.errorHandler is invoked by when an internal error occurs in the user object. If you pass the name of a user-defined function to User.errorHandler, the function will be called when this event occurs.

Compatibility

This property is compatible with scripts for Scripter .NET Client. Do not use this property in scripts for Interaction Connect.

Syntax

```
User.errorHandler(errId, ErrText)
```

Parameters

errId

 The HRESULT error number.

ErrText

 Textual description of the error.

Usage

Read Yes

Write Yes

Value Assigned

Function Pointer

 A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, define the errorId, and errorText as arguments to the function. e.g.: function foo(errorId, errorText).

Value Returned

Function Pointer

User.statusChangeHandler Callback Property

Definition

User.statusChangeHandler is invoked when the agent's status changes. This method is called when the user's status changes. If you pass the name of a user-defined function to User.statusChangeHandler, the function will be called when this event occurs.

Usage

Read Yes

Write Yes

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
User.statusChangeHandler(newStatus, untilDateTime)
```

Parameters

newStatus

>   The user's status after it changes.

untilDateTime

>   The date and/or time that the user will return, if set.

Value Assigned

Function Pointer

>   A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo". When defining your custom function, define newStatus, and untilDateTime as parameters. e.g.: function foo(newStatus, untilDateTime).

Value Returned

Function Pointer

User.userChangeHandler Callback Property

Definition

User.userChangeHandler is invoked when any of the agent's configuration attributes changes, except the agent's status. If you pass the name of a user-defined function to User.changeHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
User.changeHandler
```

Usage

Read Yes

Write Yes

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo".

> When defining your custom function, no input parameters are needed. E.g. function foo().

Value Returned

Function Pointer

User.userLoginChangeHandler Callback Property

Definition

User.userLoginChangeHandler is invoked when agent logs in or out of the system. The Login parameter will be True on a login message and False on a logout message. If you pass the name of a user-defined function to User.changeHandler, the function will be called when this event occurs.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Usage

Read Yes

Write Yes

Syntax

```
User.userLoginChangeHandler(station, login)
```

Parameters

station

> The CIC station name.

login

> True if the user has logged in, or False if the user has logged out.

Value Assigned

Function Pointer

> A function pointer is the address in memory where a user-defined function is loaded. Function pointers pass the address of a user-defined function to another function declared within an application. In a script, the function pointer is simply the name of the function. For example, if your code contains a function named "foo", the function pointer would also be named "foo".

> Specify station and login parameters when defining your custom function. E.g. function foo(station, login).

Value Returned

Function Pointer

User.canListen Property

Definition

Used to determine if the agent can listen in on calls.

Syntax

```
User.canListen
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

> Returns True if the agent's can listen in on calls.

User.canMakePrivate Property

Definition

Used to determine if the current agent's calls can be made private.

Syntax

```
User.canMakePrivate
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

> Returns True if the current agent's calls can be made private.

User.canRecord Property

Definition

Used to determine if the agent can record calls.

Syntax

```
User.canRecord
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

> Returns True if the current agent can record calls.

User.extension Property

Definition

The agent's CIC extension number.

Syntax

```
User.extension
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

String

> The current agent's CIC extension number.

User.id Property

Definition

The CIC User name of this agent.

Syntax

```
User.Id
```

Usage

Read Yes

Write Yes

Value Assigned

pVal

A CIC user ID, passed as string data.

Value Returned

String

A CIC user ID, passed as string data.

User.isDND Property

Definition

Used to determine if the agent's status is in a Do Not Disturb state.

Syntax

```
User.isDND
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

Returns True if the current agent's status is in a DND state.

User.isLoggedIn Property

Definition

Used to determine if the agent is logged in.

Syntax

`User.isLoggedIn`

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

>Returns True if the current agent is logged in.

User.isOnPhone Property

Definition

Used to determine if the agent is on the phone.

Syntax

`User.isOnPhone`

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Boolean

>Returns True if the current agent is on the phone.

User.lastError Property

Definition

This property retrieves the text of the last error that occurred in the User object. Each time a method or property is called on the User object, this value is cleared.

Compatibility

This property is compatible with scripts for Scripter .NET Client. Do not use this property in scripts for Interaction Connect.

Syntax

```
User.lastError
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

String

> The text message of the last error that occurred.

User.lastErrorId Property

Definition

This property retrieves the numeric ID of the last error that occurred in the User object. Each time a method or property is called on the User object, this value is cleared.

Compatibility

This property is compatible with scripts for Scripter .NET Client. Do not use this property in scripts for Interaction Connect.

Syntax

```
User.lastErrorId
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Integer

    The numeric ID of the last error that occurred.

User.name Property

Definition

The CIC display name of the agent.

Syntax

```
User.name
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

String

    This string contains the agent's display name.

User.statusMessage Property

Definition

This property returns the agent's status indicator (e.g. Available).

Syntax

`User.statusMessage`

Usage

Read Yes

Write No

Value Assigned

String

The user's status:
- ACD - Agent Not Answering
- At a Training Session
- At Lunch
- At Play
- Available
- Available, No ACD (Person is available for all non-ACD calls.)
- Available, Remote (Person is available on a Remote Client)
- Away From Desk
- Do Not Disturb
- Follow Up
- Gone Home
- In a meeting
- On Vacation
- Out of the Office
- Out of Town
- Working at Home

These status codes are standard with CIC. To be valid, a status code must be defined within the CIC system.

Value Returned

String

The string contains the agent's user status.

User.untilDateTime Property

Definition

Gets or sets the date and time that the agent will return from an unavailable status condition.

Syntax

```
User.untilDateTime
```

Usage

Read Yes

Write Yes

Value Assigned

Date

    A VARIANT date object representing the date-time for the unavailable status condition.

Value Returned

Date

    A VARIANT date object representing the date-time for the unavailable status condition.

User.startAccessibleQueuesEnum Enumeration Property

Definition

This property returns an enumeration of accessible queues that current agent has rights to view and modify. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Starting with 2018 R3, this property accepts an optional callback (for use with Connect scripts only) whose single parameter contains the result. See Interaction Connect Example in Queue.startCallObjectsEnum Enumeration Property for a related example that shows how a user-defined callback is used.

Syntax

```
Queue.startAccessibleQueuesEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Enumeration

     The enumeration contains the list of accessable queues.

Java Enumeration Example (JavaScript)

```javascript
// show all accessible queues for this user
var queues = scripter.myUser.startAccessibleQueuesEnum;
while (queues.hasMoreElements()) {
    alert(queues.nextElement());
}
```

User.startAvailableCampaignObjectEnum Property

Definition

This property returns an enumeration of campaigns which are available for agent to logon to or out of. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series.

The hasMoreElements method will return a Boolean true if this enumeration contains more elements. note: only campaings that the agent has view permission granted will be returned by this enumeration.

This enumeration will only return campaigns that are available to the user, and campaigns that the user has been granted view permission on.

Syntax

```
User.startAvailableCampaignObjectEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Enumeration

   The enumeration contains a list of campaign objects.

Java Enumeration Example (JavaScript)

```javascript
// show all available campaigns and their status.
var campaigns = scripter.myUser.startAvailableCampaignObjectEnum;
while (campaigns.hasMoreElements()) {
    var campaign = campaigns.nextElement();
    alert(campaign.campaignName);
    alert(campaign.campaignStateString);
}
```

User.startAvailableStatusMessagesEnum Enumeration Property

Definition

This property returns an enumeration of available status messages. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Syntax

```
Queue.startAvailableStatusMessagesEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Enumeration

The enumeration containing the list of available status messages.

Java Enumeration Example (JavaScript)

```html
<html>
<head>
<script language="javascript">
    // populate the <select> element with all valid statuses
    var Myenum = scripter.myUser.startAvailableStatusMessagesEnum;
    while (Myenum.hasMoreElements()) {
        StatusList.add(new Option(Myenum.nextElement));
    }
</script>
</head>
<body>
    <table>
        <tr>
            <td><select id="StatusList"></select></td>
        </tr>
    </table>
</body>
</html>
```

User.startLoggedInStationsEnum Enumeration Property

Definition

This property returns an enumeration of logged in stations for the current agent. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Syntax

```
Queue.startLoggedInStationsEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Java Enumeration

The enumeration contains the list of logged-in stations.

Java Enumeration Example (JavaScript)

```javascript
// show all logged-in stations for this user
var stations = scripter.myUser.startLoggedInStationsEnum;
while (stations.hasMoreElements()) {
    alert(stations.nextElement());
}
```

User.startViewableWorkgroupsEnum Enumeration Property

Definition

This property returns an enumeration of viewable workgroups (workgroups the agent has rights to view) for the current agent The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Starting with 2018 R3, this property accepts an optional callback (for use with Connect scripts only) whose single parameter contains the result. See Interaction Connect Example in Queue.startCallObjectsEnum Enumeration Property for a related example that shows how a user-defined callback is used.

Syntax

Queue.startViewableWorkgroupsEnum

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Java Enumeration

> The enumeration contains the list of viewable workgroups.

Java Enumeration Example (JavaScript)

```javascript
// show all workgroups this user is permitted to view
var workgroups = scripter.myUser.startViewableWorkgroupsEnum;
while (workgroups.hasMoreElements()) {
    alert(workgroups.nextElement());
}
```

User.startWorkgroupsEnum Enumeration Property

Definition

This property returns an enumeration of workgroups for which the agent is a member of. The enumeration is traversed using the hasMoreElements and nextElement methods of the enumeration. Each call to the nextElement method returns successive elements of the series. The hasMoreElements method will return a Boolean true if this enumeration contains more elements.

Syntax

```
Queue.startWorkgroupsEnum
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Java Enumeration

> The enumeration contains the list of member workgroups.

Java Enumeration Example (JavaScript)

```javascript
// show all workgroups this user is permitted to view
var workgroups = scripter.myUser.startViewableWorkgroupsEnum;
while (workgroups.hasMoreElements()) {
    alert(workgroups.nextElement());
}
```

dialer object

Definition

The dialer object encapsulates properties of the agent's session with Dialer and properties such as which campaigns the agent is active in.

Methods

dialer.endBreak	Ends a break so that the agent will receive campaign calls.
dialer.requestBreak	This method requests a break. Dialer will automatically grant the break when its predictive algorithm determines that enough agents are available to process calls without this agent.
dialer.sendCustomHandlerNotification	Initiates custom handlers through the custom notification initiator.
dialer.subscribeToCustomHandlerNotification	This hook allows a custom script to asynchronously subscribe and listen to a Send custom notification step in a custom handler.

Callbacks

campaignLoginHandler	This handler is invoked when agents are logged into a campaign by an administrator.
campaignLogoutHandler	This handler is invoked when agents log out of a campaign.
breakRequestedHandler	This handler is invoked whenever the agent requests a break.
breakGrantedHandler	This handler is invoked whenever a break request is granted.
breakEndedHandler	This handler is invoked whenever a break ends.
campaignsChangedHandler	This handler is invoked whenever a user is logged into or out of a campaign.

Properties

breakStatus	This property returns the agent's break status (On Break, Break Pending, Not on Break).
campaigns	This property returns an array of campaigns that the agent is active in.

Example

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Dialer Object Demo Page</title>
    <script src="http://code.jquery.com/jquery-1.9.1.js"></script>
    <script language="javascript">

        // This is a sample script utilizing the dialer object in an autoloaded page.
        $(document).ready(function() {
            init();
        });
        function init() {
            // Add a handler to pick up changes to the campaigns list. (new campaings being logged into, or
out of)
            scripter.dialer.campaignsChangedHandler = handleDialerChange;
            InitTagValues()
        }
        function InitTagValues() {
            $("#tagAgentId").html(scripter.myUser.name);
            $("#tagServerId").html(scripter.notifierName);
            $("#tagFirstCampaignName").html(scripter.dialer.firstCampaignName);
            $("#tagCampaignEnum").html(fillCampaignList());
            $("#tagDialerCampaigns").text(scripter.dialer.Campaigns);
            $("#tagBreakStatus").html(evalBreakEnum(scripter.dialer.breakStatus));
            $("#RequestBreakBtn").click(
                function() {
                    scripter.dialer.requestBreak();
                });
            $("#EndBreakBtn").click(
                function() {
                    scripter.dialer.endBreak();
                });
            $("#RefreshBtn").click(
                function() {
                    window.location = "http://localhost/script.htm";
                }
            );
            scripter.dialer.breakRequestedHandler = HandleBreakChange;
            scripter.dialer.breakGrantedHandler = HandleBreakChange;
            scripter.dialer.breakEndedHandler = HandleBreakChange;
            scripter.dialer.campaignLoginHandler = handleLogInChange;
```

```javascript
                scripter.dialer.campaignLogoutHandler = handleLogoutChange;
            }
            function HandleBreakChange() {
                $("#tagBreakStatus").html(evalBreakEnum(scripter.dialer.breakStatus));
            }
            function handleDialerChange() {
                InitTagValues();
            }
            function handleLogoutChange(campaignName) {
                alert(campaignName);
                InitTagValues()
            }
            function handleLogInChange(campaignName) {
                alert(campaignName);
                InitTagValues();
            }
            function evalBreakEnum(e) {
                switch (e) {
                    case 0:
                        return "Not On Break";
                        break;
                    case 1:
                        return "Break Pending";
                        break;
                    case 2:
                        return "On Break";
                        break;
                    case 3:
                        return "Not Logged Into a Campaign";
                        break;
                }
            }
            function fillCampaignList() {
                var campaigns = scripter.myUser.startAvailableCampaignObjectEnum;
                var campaignList = '';
                while (campaigns.hasMoreElements()) {
                    var campaign = campaigns.nextElement();
                    campaignList += campaign.campaignName + ',';
                }
                return campaignList;
            }
    </script>
</head>
<body>
    <button id="RefreshBtn">Text</button><br />
    <table border="1" cellpadding="5">
        <tr>
            <td>Agent ID</td>
            <td><span id="tagAgentId" /></td>
        </tr>
        <tr>
            <td>scripter.notifierName</td>
            <td><span id="tagServerId" /></td>
        </tr>
        <tr>
            <td>scripter.dialer.firstCampaignName</td>
            <td><span id="tagFirstCampaignName" /></td>
        </tr>
        <tr>
            <td>scripter.myUser.startAvailableCampaignObjectEnum</td>
            <td><span id="tagCampaignEnum" /></td>
        </tr>
        <tr>
            <td>scripter.dialer.Campaigns;</td>
            <td><span id="tagDialerCampaigns" /></td>
        </tr>
        <tr>
            <td>scripter.dialer.breakStatus</td>
            <td><span id="tagBreakStatus" /></td>
        </tr>
```

```html
        <tr>
            <td><button id="RequestBreakBtn">request break</button></td>
            <td><button id="EndBreakBtn">end break</button></td>
        </tr>
    </table>
</body>
</html>
```

dialer.endBreak Method

Definition

Ends a break so that the agent will receive campaign calls.

Syntax

```
dialer.endBreak();
```

Prototype

```
dialer.endBreak()
```

Input Parameters

None.

Example

```javascript
$("#RequestBreakBtn").click(
    function() {
        scripter.dialer.endBreak();
    }
);
```

dialer.requestBreak Method

Definition

This method requests a break. Dialer will authmatically grant the break when its predictive algorithm determines that enough agents are available to process calls without this agent.

Syntax

```
dialer.requestBreak ();
```

Prototype

```
dialer.requestBreak()
```

Input Parameters

None.

Example

```
$("#RequestBreakBtn").click(
    function() {
        scripter.dialer.requestBreak();
    }
);
```

dialer.sendCustomHandlerNotification Method

Definition

This method allows a custom script to initiate custom handlers through the custom notification initiator. Use with care. Starting a very complex, long-running handler could potentially affect the performance of a PureConnect server.

> **NOTE:** This functionality is only available in Interaction Connect. It is not available in Scripter .NET.

Input Parameters

ObjectId

> The Object Id defined in the handler's initiator.

EventId

> The Notification Event defined in the handler's initiator.

dataArray

> All other data is passed by an array of string values.

Value Returned

There is no return value. This call takes place asynchronously.

Example

The example calls the sendCustomHandlerNotification method from a static dialer object.

```
var dataArray = ['someInformation', 'moreInfo', 'lastInfo'];
scripter.dialer.sendCustomHandlerNotification('sampleObjectID', 'sampleEventID', dataArray);
```

The sampleObjectID and sampleEventID should correspond to the Object ID and Notification Event properties defined in the initiator that the user wants to initiate. All other user defined data can be passed as an array of strings(eg: dataArray).

> **NOTE:** A custom script can also subscribe to a custom handler notification to listen for responses after initiating the custom handler. See dialer.subscribeToCustomHandlerNotification Method.

dialer.subscribeToCustomHandlerNotification Method

Definition

This hook allows a custom script to asynchronously subscribe and listen to a Send custom notification step in a custom handler. To do this, call the subscribeToCustomHandlerNotification method from a static dialer object.

> **NOTE:** This functionality is only available in Interaction Connect. It is not available in Scripter .NET.

Input Parameters

headers

> The scripter has to define the header containing the objectID, eventID pairs that they want to subscribe to. For example:

```
var headers = [
  { "objectId": "CompletePurchaseObjectID", "eventId": "CompletePurchaseEventID" },
  { "objectId": "NextTransactionObjectID", "eventId": "NextTransactionEventID" }
];
```

responseCallBackHandler

> This is a user-defined function that is called upon the reception of a notification from the custom handler. This function accepts three parameters: `objectId`, `eventId` and `data`. For example:

```
function executeWhenHandlerResponds(objectId, eventId, data)
{ console.log('ObjectID returned was: '+ objectId); console.log('EventID returned was: '+ eventId);
console.log('DataReturned was '+ data); }
```

> The user can parse the data object returned as they wish. However, they should be able to anticipate the type of data the notification contains which requires knowledge of the custom handler's steps.

Value Returned

There is no return value.

> **NOTE:** Upon subscription to the SendCustomNotification step, a script may initiate custom handlers through the sendCustomHandlerNotification from the static dialer object. See [dialer.sendCustomHandlerNotification Method](#).

dialer.campaignLoginHandler Callback Property

Definition

dialer.campaignLoginHander is invoked when agents are logged into a campaign by an administrator.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
dialer.campaignLoginHandler (campaignName)
```

Parameters

campaignName

   The name of the campaign the agent was logged into.

Example

```
scripter.dialer.campaignLoginHandler = handleLogInChange;
function handleLogInChange(campaignName) {
    alert(campaignName);
}
```

dialer.campaignLogoutHandler Callback Property

Definition

dialer.campaignLogoutHander is invoked when agents log out of a campaign.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
Dialer.campaignLogoutHandler (campaignName)
```

Parameters

campaignName

   The name of the campaign the agent was logged out of.

Example

```
scripter.dialer.campaignLogoutHandler = handleLogoutChange;
function handleLogoutChange(campaignName) {
    alert(campaignName);
}
```

dialer.breakRequestedHandler Callback Property

Definition

dialer.breakRequestedHandler is invoked whenever the agent requests a break.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
dialer.breakRequestedHandler ()
```

Parameters

None

Example

```
scripter.dialer.breakRequestedHandler = HandleBreakChange;
function HandleBreakChange() {
    alert(scripter.dialer.breakStatus));
}
```

dialer.breakGrantedHandler Callback Property

Definition

dialer.breakGrantedHandler is invoked whenever a break request is granted.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
dialer.breakGrantedHandler ()
```

Parameters

None.

dialer.breakEndedHandler Callback Property

Definition

dialer.breakEndedHandler is invoked whenever a break ends.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
dialer.breakEndedHandler ()
```

Parameters

None.

campaignsChangedHandler Callback Property

Definition

dialer.campaignsChangedHandler is invoked whenever a user is logged into or out of a campaign.

Compatibility

This callback is compatible with scripts for Scripter .NET Client or Interaction Connect.

Syntax

```
dialer.campaignsChangedHandler ()
```

Parameters

None.

Example

See also: dialer object sample script

```
scripter.dialer.campaignsChangedHandler = handleDialerChange;
function handleDialerChange() {
    InitTagValues();
}
```

dialer.breakStatus Property

Definition

This property returns the agent's break status (On Break, Break Pending, Not on Break).

Syntax

```
dialer.breakStatus
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

breakStatus

Possible return values:

0	NotOnBreak
1	BreakPending
2	OnBreak
3	NotLoggedIn

Example

```html
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Dialer Object Demo Page</title>
    <script src="http://code.jquery.com/jquery-1.9.1.js"></script>
    <script language="javascript">
        $(document).ready(function() {
            init();
        });
        function init() {
            $("#tagBreakStatus").html(evalBreakEnum(scripter.dialer.breakStatus));
        }
        function evalBreakEnum(e) {
            switch (e) {
                case 0:
                    return "Not On Break";
                    break;
                case 1:
                    return "Break Pending";
                    break;
                case 2:
                    return "On Break";
                    break;
                case 3:
                    return "Not Logged Into a Campaign";
                    break;
            }
        }
    </script>
</head>
<body>
    <span id="tagBreakStatus" />
</body>
</html>
```

dialer.campaigns Property

Definition

This property returns an array of campaigns that the agent is active in.

Syntax

```
dialer.campaigns
```

Usage

Read Yes

Write No

Value Assigned

None.

Value Returned

Campaigns

JavaScript array of campaign name strings.

Example

```
$("#tagDialerCampaigns").text(scripter.dialer.Campaigns);
```

Script Examples

This section discusses commonly scripted programming tasks in Advanced Interaction Scripter, including:

- Agent Breaks
- Blended Campaigns
- Blind Transfer
- Conference Calls
- Consult Transfer
- Consult Transfer with Disposition
- Get and Set attributes
- Inbound Waiting for Call Page
- Play Digits to a Call
- Preview Campaigns
- Supporting Finishing Agents
- Transferring Calls
- User Queue Watcher Script
- Workgroup Queue Watcher Script

Agent Breaks

Typically, it is desirable to allow agents to take a break through their script. A simple way to do this is to give the agent a button to request the break and react accordingly when the break is granted. The following Scripter functions will be used:

- IS_Action_RequestBreak
- IS_Action_EndBreak
- IS_Action_ClientStatus
- IS_Event_BreakGranted

By clicking on a button, an agent might request a break. However, the break is not necessarily granted immediately. If there are not enough available agents to handle the calls already dialed or if the agent has not yet dispositioned the current call, then Dialer will not grant the break immediately.

When Dialer grants the break, it fires an event. We must write a function to handle this break-granted event. When the agent goes on break we should change their status to an unavailable status. In addition, we might choose to do various other tasks such as providing an 'End Break' button or providing a break timer. Because the break-granted event might not be fired immediately, we must be aware of when it could potentially be fired. Breaks are always granted immediately after a disposition. The following are appropriate pages on which to handle the break-granted event:

- Any page that dispositions a call.
- The call waiting page.

The script should allow an agent to easily return from break. When returning from break, the script should return the agent to the available agents pool for Dialer, set the agent status to available, and display the call waiting page (typically where the agent's status is set to available). Once the break cycle is completed, the agent returns to producing results through the running campaigns.

Break Process Flowcharts

Request Break

Request break

No
Break granted?

Yes

Update script for agent on break

Return from Break

Request end break

Return agent to Dialer available agents

Goto call waiting page
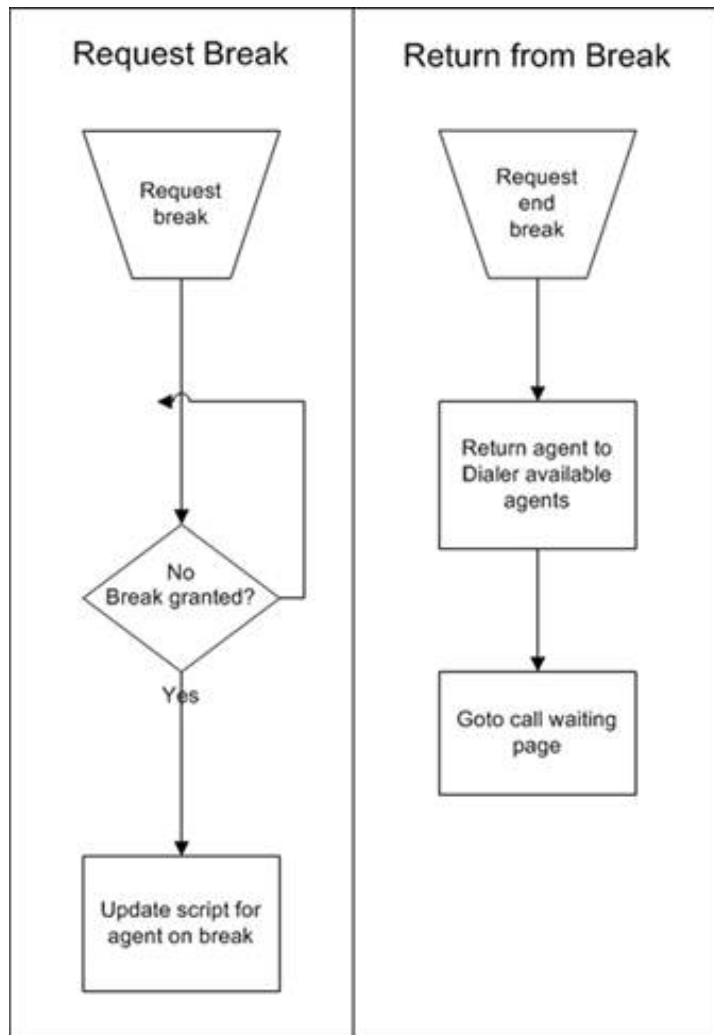
Break Process Pseudocode

Break request

When a 'break request' button is pressed:

    Send a break request

End function

Break-granted event

When the break is granted:

    Set the agent's status to unavailable

    Go to the designated break page

End function

Return from break

When agent ends the break:

    Return the agent to the Dialer available agents pool

    Go to the call-waiting page

End function

Break Process JavaScript

Break request

```javascript
function RequestBreak() {
    // Send a break request
    IS_Action_RequestBreak.click();
}
```

Break-granted event

```javascript
function IS_Event_BreakGranted() {
    // Set the agent's status to unavailable
    IS_Action_ClientStatus.statuskey = 'On Break';
    IS_Action_ClientStatus.click();
    // Goto the designated break page
    location.href = 'break.htm';
}
```

Remember that the break-granted event should appear on any page in which a break might be granted -- regardless where it was requested. This includes:

- Any page that dispositions a call
- The call waiting page

Return from break

```javascript
function EndBreak() {
    // Return the agent to the Dialer available agents pool
    IS_Action_EndBreak.click();
    // Goto the call-waiting page. This page will also set the agent status to available
    location.href = 'index.htm';
}
```

Blended Campaigns

When scripting blended campaigns, there will be multiple scripts. One script will handle inbound calls, but a separate script will handle the Dialer generated calls. The inbound script will not be launched by logging into a campaign. Rather the inbound script will either be launched manually or through an Interaction Scripter command line parameter. Use the /autologin switch to tell Interaction Scripter which script to launch at run time (as opposed to login time). /autologin may be used multiple times in one target path to launch multiple scripts.

Because the inbound script will not be working with Dialer generated calls, the IS_Actions will not allow for call control. Instead, the scripter object allows for manipulation of non-dialer calls, as well as queue management. The following Scripter functions will be used:

- scripter.MyQueue.objectAddedHandler
- scripter.CallObject.getAttribute
- scripter.CallObject.pickup
- scripter.CallObject.disconnect

When the inbound script loads, the scripter.MyQueue.objectAddedHandler function must be initialized. Simply assign the function to run when an object is added to "MyQueue" (from the perspective of the agent). In addition, two parameters are passed to the objectAddedHandler:

- TypeId - a numeric identifier of the type of object that entered the queue
- ObjectId - the unique id of the object

Once an object comes into the agent's queue, determine if that object is an inbound call. An ObjectId of 2 indicates that the object is an inbound call. Assign that object id to the scripter call object. This assignment will be valid throughout the lifespan of that scripter object.

Certainly, a script should not be associated with a family call or a lunch invitation from a coworker. In fact, the same script might not be associated for all work calls. If the object is an inbound call, then we need a mechanism by which to determine whether to associate a script with that call. This will be done by looking at a custom call attribute or through selective workgroup membership, similar to what was done for a finishing agent. In this way, a different script can be popped according to the specific workgroup or some other attribute linked to the call.

Once a decision has been made about which script to pop, but prior to actually moving to that script, it is important to reset the object added handler. Do so by setting its value to null. The conclusion of the inbound script should return to this decision making page. At that time, the object added handler will be reinitialized. If the script should not continue with this call, then reset the scripter call object's id. Set it to -1 to designate that it is not currently assigned to any call object. As the script continues, remember to use scripter call object functions, not IS_Action functions, to work with the interaction. In addition, recognize that there is not a Dialer record explicitly associated with this call. Therefore, any IS_Attr functions will not be available for database functions.

Be careful. It is possible to handle inbound and outbound calls through the same script. However, for the script to handle the Dialer generated outbound calls, it must be launched through the campaign login mechanism. Consequently, the script will be closed when the campaign is logged off or stops. Logging off or stopping a campaign would terminate the script, thus it could no longer handle inbound calls.

Blended Campaign Flowcharts

Initialize Object Added Handler

Assign ObjectAdded function to scripter.MyQueue. objectAddedHandler

Handle Incoming Call

Object Added To Agent Queue

↓

If Object Is Call —No→ Do Nothing

↓ Yes

Assign Object Id To Scripter Call Object

↓

Get Workgroup Attribute Of Call Object

↓ Yes

If Workgroup Is Dialer Workgroup —No→ Release Call Object

↓ Yes

Reset Object Added Handler

↓

Pop Script

Blended Campaign Pseudocode

Initialize object added handler:

```
When the script loads:
  Run the ObjectAdded function when an object is added to the agent queue.
End function
```

Handle incoming call:

```
When an object is added to the agent queue:
  If the object is a call object:
    Assign object id to this scripter object
    If the call object is associated with the Dialer workgroup:
      Reset the object added handler
      Launch the inbound call script
    If the call object is not associated with the Dialer workgroup:
  Release the object
End function
```

Blended Campaign JavaScript

Initialize object added handler

```
window.onload = Init;
function Init() {
    // queue.objectAddedHandler specifies a procedure to // be called when an object is added to a queue
    // syntax: Queue.objectAddedHandler(TypeId,ObjectId)
    scripter.myQueue.objectAddedHandler = ObjectAdded;
}
```

Handle incoming call

```
// Object added handler function: object type and object
// id are passed implicitly
function ObjectAdded(ObjType, ObjId) {
    // ObjType == 2 are call objects
    if (ObjType == 2) {
        // Assign id of the added object to this object
        // This assignment will persist for the duration
        // of the script interaction.
        scripter.callObject.id = ObjId;
        var objCall = scripter.createCallObject();
        objCall.id = ObjId;
        // EIC_Workgroup - legacy naming.  Attributes often start with EIC_
        var workgroup = objCall.getAttribute("EIC_Workgroup");
        // Workgroup queue name is case sensitive.
        // If object is assigned to VOYAGE workgroup
        if (workgroup == "Workgroup Queue:VOYAGE") {
            // Reset the object added handler
            scripter.myQueue.ObjectAddedHandler = null;
            // Launch inbound call script
            location.href = "inboundopen.htm";
        } else {
            // If interaction is not a VOYAGE call
            // release the call object
            scripter.callObject.id = -1;
        }
    }
}
```

Blind Transfer

The script below demonstrates how to perform a blind transfer using the scripter advanced API.

```html
<html>
<head>
    <title>Blind Transfer Example</title>
    <script type="text/javascript" defer>
        function BlindTransfer(RemoteNumber1, RemoteNumber2) {
            // call party 1
            scripter.callObject.dial(RemoteNumber1, false);
            var doConnect = window.confirm("Blind transfer to party 2?");
            if (!doConnect) {
                scripter.callObject.disconnect();
                return;
            }
            // make the transfer
            scripter.callObject.blindTransfer(RemoteNumber2);
        }
    </script>
</head>
<body>
<p>Party 1 <input id=RemNum1>  Party 2 <input id=RemNum2></p>
<p><input type=button value="Blind Transfer" onclick=BlindTransfer(RemNum1.value,RemNum2.value)></p>
</body>
</html>
```

Conference Calls

Conference calls allow many parties to participate in an interaction simultaneously. This is useful when it is necessary to connect multiple agents to an interaction or pull a supervisor into a conversation. Creating and managing a conference call in Scripter is surprisingly simple. The following Scripter functions will be used:

- scripter.conferenceObject.create
- scripter.conferenceObject.add
- scripter.callObject.dial
- scripter.callObject.disconnect

As with any interaction, knowing the CallID opens up many possibilities. Building conference calls is no different. It is simply a matter of adding objects, by their CallID, to a conference call object. Likewise, it is important to manage CallIDs by resetting them, simply by assigning their value to -1, before creating new conference or call objects. If a conference has not already been created then that will be the first step. After that, create a call object by dialing a party and then use the CallID of that object to add it to a conference. The scripter.conferenceObject.add method will add already created call objects to a conference.

In the example javascript code provided, the conference create and add processes are included in the same function. Use a global boolean variable to make a decision whether a new conference must be created.

Conference Call Process Flowchart

```
   ┌─────────────┐
   │ Button - Add│
   │   Call To   │
   │ Conference  │
   └─────────────┘
         │
         ▼
      ╱If New╲        Yes    ┌──────────────┐   ┌──────────────┐   ┌──────────────┐   ┌──────────────┐
     ╱Conference╲─────────▶  │Assign Dialer │──▶│Reset Any     │──▶│Create        │──▶│Set Flag To   │
     ╲          ╱            │Call To       │   │Existing      │   │Conference    │   │Indiacate     │
      ╲        ╱             │Scripter Call │   │Conference    │   │Call With     │   │Conference    │
         │                   │Object        │   │Object        │   │Dialer Call   │   │Created       │
         │ No                └──────────────┘   └──────────────┘   └──────────────┘   └──────────────┘
         ▼
   ┌─────────────┐
   │Reset Scripter│
   │Call Object   │
   └─────────────┘
         │
         ▼
   ┌─────────────┐
   │Dial Third   │
   │Party        │
   └─────────────┘
         │
         ▼
   ┌─────────────┐
   │Prompt To    │
   │Transfer     │
   └─────────────┘
         │
         ▼
      ╱If Transfer╲   No    ┌──────────────┐    ┌──────────────┐
     ╱            ╲────────▶│Disconnect    │───▶│ End Function │
     ╲            ╱         │Third Party   │    └──────────────┘
         │                  └──────────────┘
         │ Yes
         ▼
   ┌─────────────┐
   │Add Third    │
   │Party To     │
   │Conference   │
   └─────────────┘
         │
         ▼
   ┌──────────────┐
   │ End Function │
   └──────────────┘
```

Conference Call Process Pseudocode

```
Global flag to indicate whether this is a new or existing conference
When add to conference button is pressed:
    If this is a new conference
      Assign the Dialer call to the scripter call object
      Reset any existing conference object
      Create the conference with the Dialer call
      Set flag to indicate conference created
    Reset the scripter call object
    Dial the third party
    Prompt to transfer
    If no:
      Disconnect third party
    End function
    Add third party to conference call
End function
```

Conference Call Process Javascript

There are several Javascript examples of the Conference Call Process:
- Conference Call Process Javascript
- Create Conference Call and add parties
- Conference with Third Party
- Conference with Third Party then Locally Disconnect
- Call Third Party, Create Conference, then Disconnect Agent

Conference Call Process JavaScript

```javascript
// Global flag to indicate whether this is a new or existing conference
var CreatedConference = 0;
function AddToConference(p_RemoteNum) {
    // If this is a new conference
    if (CreatedConference == 0) {

        // Assign the Dialer call to the scripter call object
        scripter.callObject.id = IS_Attr_CallID.value;
        // Reset any existing conference object
        scripter.conferenceObject.id = -1

        // Create the conference with the Dialer call
        scripter.conferenceObject.create(scripter.callObject);
        // Set flag to indicate conference created
        CreatedConference = 1;
    }
    // Reset the scripter call object
    scripter.callObject.id = -1;

    // Dial the third party
    scripter.callObject.dial(p_RemoteNum, false);

    // Prompt to transfer
    var doConnect = window.confirm("Add party to conference?");

    // If no then disconnect third party and end function
    if (!doConnect) {
        alert("Disconnecting");
        scripter.callObject.disconnect();
        return;

        // Add third party to conference call}
        scripter.conferenceObject.add(scripter.callObject);
    }
}
```

Create Conference Call and add parties

The JavaScript below demonstrates how to create a conference call and add parties to the conference call.

```html
<html>
<head>
    <title>Conference Example</title>
    <script type="text/javascript" defer>
        function AddToConference(RemoteNumber) {
            // contact the specified party
            // this will automatically put current call on hold
            scripter.callObject.id = -1;
            scripter.callObject.dial(RemoteNumber, false);
            var doConnect = window.confirm("Add party to conference?");
            if (doConnect != "yes") {
                // do not add the party to the conference
                scripter.callObject.disconnect();
                return;
            }
            // create or add to a conference
            if (scripter.conferenceObject.id == 0xffffffff) {
                scripter.conferenceObject.create(scripter.callObject);
            } else {
                scripter.conferenceObject.add(scripter.callObject);
            }
        }
    </script>
</head>
<body>
    <p><input type=button value="Add to Conference" onclick=AddToConference(RemNum.value)>  
        <input id=RemNum></p>
</body>
</html>
```

Conference with Third Party

The JavaScript function below demonstrates how an agent can call a third party, consult with that third. Then bring the third party in with the customer and the agent. Then once the verification process is complete, disconnect the third party, and continue with the call with the customer and the agent.

```javascript
function ConferenceWithThirdParty(p_Number) {

    // create a callobject
    var p_mCallObj = scripter.createCallObject();

    // create a conference object
    var p_ConferenceObject = scripter.createConferenceObject();
    var iRes1 = confirm("Would you like to call the 3rd party?");

    // user selected OK, so let's call the 3rd party
    if (iRes1) {
        p_mCallObj.dial(p_Number, false);
        var iRes2 = confirm("Press OK when you are ready to conference in the 3rd party");

        // conference in the third party
        if (iRes2) {
            scripter.callObject.id = IS_ATTR_Callid.value;
            // create the conference object
            // add 3rd party
            p_ConferenceObject.Create(p_mCallObj);

            // pick up the call, it is probably on hold
            scripter.callObject.pickup();
            // add scripter call
            p_ConferenceObject.add(scripter.callObject);
            // pickup 3rd party just in case
            p_mCallObj.pickup();                    var iRes3 = confirm("Press OK when you are ready to disconnect
the  third party call");
            if (iRes3) {
```

```
                   DisconnectCallInConference(p_ConferenceObject.id, p_Number);
            }
        } else {
            // they did not want to transfer, so lets disconnect the 3rd party call and
            // pick up the original call
            // disconnect 3rd party call
            p_mCallObj.disconnect();
            // pick up original call
            scripter.callObject.id = IS_ATTR_Callid.value;
            scripter.callObject.pickup();
        }
    }
}
/**
 * This method is a utility function that will be used to disconnect a specific leg
 of the *conference call where the RemoteTn(Number Dialed)
 * is equal to the number passed in.
 * @param pConferenceID - id of the conference to search
 * @param pDialedNumber - number dialed that needs to be disconnected
 * @Author Customer Care
 */
function DisconnectCallInConference(pConferenceID, pDialedNumber) {
    // create conference object
    var ConfObj = scripter.createConferenceObject();

    // set id of the conference object to the one created when the conference was created
    ConfObj.id = pConferenceID;
    // get handle to enumerator for collection
    var _enum = ConfObj.startMemberIdsEnum;

    // enumerate conference object collection
    while (_enum.hasMoreElements()) {

        // grab id of call object
        var callObjId = _enum.nextElement();

        // create call object
        var callObj = scripter.createCallObject();

        // assign id to call object
        callObj.id = callObjId;
        // check the remoteTn value of the call, if equal then that is the call to disconnect
        if (callObj.getAttribute('Eic_RemoteTn') == pDialedNumber) {
            ConfObj.disconnectParty(callObj.id);
        }
    }
}
```

Conference with Third Party then Locally Disconnect

The JavaScript function below demonstrates how an agent can call a third party, consult with that third. Then bring the third party in with the customer and the agent, then take themselves out of the conference, this way the third party is talking with the customer and the agent is free to take other calls.

```javascript
function ConferenceWithThirdPartyLocalDisconnect(p_Number) {
    var p_mCallObj = scripter.createCallObject();
    var p_ConferenceObject = scripter.createConferenceObject();
    var iRes1 = confirm("Would you like to call the 3rd party?");

    // user selected OK, so let's call the 3rd party
    if (iRes1) {
        p_mCallObj.dial(p_Number, false);
        var iRes2 = confirm("Press OK when you are ready to conference in the 3rd party");

        // conference in the third party
        if (iRes2) {
            scripter.callObject.id = IS_ATTR_Callid.value;
            // create the conference object
            // add 3rd party
            p_ConferenceObject.Create(p_mCallObj);
            // pick up the call, it is probably on hold
            scripter.callObject.pickup();
            // add scripter call
            p_ConferenceObject.add(scripter.callObject);
            // pickup 3rd party just in case
            p_mCallObj.pickup();
            var iRes3 = confirm("Press OK when you are ready to disconnect from this call");
            if (iRes3) {
                scripter.callObject.disconnect();
            }
        } else {
            // they did not want to transfer, so lets disconnect the 3rd party call and pick up the original
call
            // disconnect 3rd party call
            p_mCallObj.disconnect();
            // pick up original call
            scripter.callObject.id = IS_ATTR_Callid.value;
            scripter.callObject.pickup();
        }
    }
}
```

Call Third Party, Create Conference, then Disconnect Agent

The JavaScript function below demonstrates how to call a third party, then create a conference with the customer and the third party, then disconnect the agent leg of the call while dispositioning the call before the transfer and letting the agent be ready for another call. The key in this function is to disposition before transfer to avoid the scripter warning that a call was removed from the users' queue without being completed.

```javascript
/*
This utility function will return the call object that connects the current user
to the specified conference
* @param conferenceId - The id of a conference the current user is a part of
* @return - The call object that is the user's leg of the conference, or null if
the call leg could not be found
*/
function FindConferenceLeg(conferenceId) {
    var callObjects = scripter.myQueue.startCallObjectsEnum;
    while (callObjects.hasMoreElements()) {
        var callObject = callObjects.nextElement();
        if (callObject.conferenceId != conferenceId) {
            continue;
        }
        if (callObject.state != 105) {
            continue;
        }
```

```javascript
        return callObject;
    }
    return null;
}
/*
This utility function will disconnect the current user from a specified conference,
leaving the other parties in the conference connected.
* @param conferenceObject - A conference object the current user is a part of
*/
function DisconnectFromConference(conferenceObject) {
    var conferenceLeg = FindConferenceLeg(conferenceObject.id);
    if (conferenceLeg != null) {
        conferenceObject.disconnectParty(conferenceLeg.id);
    }

}
/**
This method takes care of performing a consult transfer and dispositioning the call before transfer
* @param p_Number - the number to dial for the 3rd party
* @param p_WrapupCode - the wrap-up code to use when dispositioning the call
* @param p_Page - the page to navigate to when done with the call
*/
function Conference3rdPartyTransferWithDisposition(p_Number, p_WrapupCode, p_page) {
    var p_mCallObj = scripter.createCallObject();
    var p_ConferenceObject = scripter.createConferenceObject();
    var iRes1 = confirm("Would you like to call the 3rd party?");
    // user selected OK, so let's call the 3rd party
    if (iRes1) {
        p_mCallObj.dial(p_Number, false);
        var iRes2 = confirm("Press OK when you are ready to conference in the 3rd party");
        // conference in the third party
        if (iRes2) {
            scripter.callObject.id = IS_ATTR_Callid.value;
            // create the conference object
            // add 3rd party
            p_ConferenceObject.Create(p_mCallObj);
            // pick up the call, it is probably on hold
            scripter.callObject.pickup();
            // add scripter call
            p_ConferenceObject.add(scripter.callObject);
            //pickup 3rd party just in case
            p_mCallObj.pickup();
            var iRes3 = confirm("Press OK when you are ready to disconnect from this call");
            if (iRes3) {
                // issue the disposition
                // since it is ok to transfer the call, disposition it before the transfer
                IS_Action_CallComplete.wrapupcode = p_WrapupCode;
                IS_Action_CallComplete.click();
                // disconnect from the conference
                DisconnectFromConference(p_ConferenceObject);
                // if we have a page parameter, then use it
                if (p_page != null)
                    location.href = p_page;
            }
        } else {
            // they did not want to transfer, so lets disconnect the 3rd party call and pick up the original
call
            // disconnect 3rd party call
            p_mCallObj.disconnect();
            // pick up original call
            scripter.callObject.id = IS_ATTR_Callid.value;
            scripter.callObject.pickup();
        }
    }
}
```

Consult Transfer

The JavaScript below demonstrates how to perform a consult transfer.

```html
<html>
<head>
    <title>Consult Transfer Example</title>
    <script type="text/javascript" defer>
        function ConsultTransfer(RemoteNumber1, RemoteNumber2) {

            // call party 1
            var callObject1 = scripter.createCallObject();
            callObject1.dial(RemoteNumber1, false);
            var doConnect = window.confirm("Call party 2?");
            if (!doConnect) {
                callObject1.disconnect();
                return;
            }
            // call party 2
            // this automatically puts party 1 on hold
            var callObject2 = scripter.createCallObject();
            callObject2.dial(RemoteNumber2, false);
            var doTransfer = window.confirm("Transfer party 1 to party 2?");
            if (!doTransfer) {
                callObject2.disconnect();
                return;
            }
            // make the transfer
            callObject1.consultTransfer(callObject2.id);
                       // free objects
            callObject1.id = -1;
            callObject2.id = -1;
        }
    </script>
</head>
<body>
    <p>Party 1 <input id=RemNum1>  Party 2 <input id=RemNum2></p>
    <p><input type=button value="Consult Transfer" onclick=ConsultTransfer(RemNum1.value,RemNum2.value)></p>
</body>
</html>
```

Consult Transfer with Disposition

The JavaScript function below demonstrates how to consult transfer the customer to a third party while dispositioning the call before the transfer and letting the agent be ready for another call. The key in this function is to disposition before transfer to avoid the scripter warning that a call was removed from the user's queue without being completed.

```javascript
/**
This method takes care of performing a consult transfer and dispositioning the call before transfer
* @param p_Number - the number to dial for the 3rd party
* @param p_WrapupCode - the wrap-up code to use when dispositioning the call
*/
function ConsultTransferWithDisposition(p_Number, p_WrapupCode) {
    var p_mCallObj = scripter.createCallObject();
    var iRes1 = confirm("Would you like to call the 3rd party?");
    // user selected OK, so let's call the 3rd party
    if (iRes1){
        p_mCallObj.dial(p_Number, false);
                //set up the consult transfer
        IS_Action_Transfer.consult = true;

        //set up the recipient call object
        IS_Action_Transfer.recipient = p_mCallObj.id;

        var iRes2 = confirm("Press OK when you are ready to transfer the call");
        // Transfer call to third party
            if (iRes2) {
              // since it is ok to transfer the call, disposition it before the transfer

                IS_Action_CallComplete.wrapupcode = p_WrapupCode;
                IS_Action_CallComplete.click();
                scripter.callObject.id = IS_ATTR_Callid.value;
                // pick up the call, it is probably on hold
                scripter.callObject.pickup();
                // now execute the consult transfer that has been set up
                IS_Action_Transfer.click();
            }
            else {
              // they did not want to transfer, so lets disconnect the 3rd party call and pick up the original
call

                // disconnect 3rd party call
                 p_mCallObj.disconnect();
                 // pick up original call
                scripter.callObject.id = IS_ATTR_Callid.value;
                scripter.callObject.pickup();
            }
        }
    }
}
```

Get and Set attributes

The functions below demonstrate how to set and get call attributes from a call that is on the user's queue while logged into Scripter. It also demonstrates how useful the IS_Action_Trace function is.

```javascript
/*
This method will return the value of a given call attribute
* @param p_sAttributeName
* @Author PureConnect Customer Care
*/
function getCallAttr(p_sAttributeName) {
    var CallObj = scripter.createCallObject();
    var attrValue = "";
    CallObj.id = IS_ATTR_Callid.value;
    if (CallObj.id != -1) {
        attrValue = CallObj.getAttribute(p_sAttributeName);
        IS_TraceNote('Callid: ' + CallObj.id + ' Getting Attr: ' + p_sAttributeName +
            "=" + attrValue);
        return attrValue;
    }
    else {
        IS_TraceError('Could not get valid call object');
        return attrValue;
    }
}
/*
This method will set the value of a given call attribute
* @param p_sAttributeName
* @param p_sAttributeValue
* @Author PureConnect Customer Care
*/
function setCallAttr(p_sAttributeName, p_sAttributeValue) {
    var CallObj = scripter.createCallObject();
    var attrValue = "";
    CallObj.id = IS_ATTR_Callid.value;
    if (CallObj.id != -1) {
        IS_TraceNote('Callid: ' + CallObj.id + ' Setting Attr: ' + p_sAttributeName + "=" +
p_sAttributeValue);
        CallObj.setAttribute(p_sAttributeName, p_sAttributeValue);
    }
    else {
        IS_TraceError('Could not get valid call object');
    }
}
/*
 * This method will bring send custom trace messages to the Scripter vwrlog
 * @Author PureConnect Customer Care*/
function IS_TraceNote(p_message) {
    IS_Action_Trace.message = p_message;
    IS_Action_Trace.level = 3; // 3= Notes level

    IS_Action_Trace.click();
}
function IS_TraceError(p_message) {
    IS_Action_Trace.message = p_message;
    IS_Action_Trace.level = 0; //0 = Error level

    IS_Action_Trace.click();
}
```

Inbound Waiting For Call Page

This JavaScript code demonstrates how to set up a waiting for call page for an inbound workgroup. It also demonstrates how a single page can handle calls from different inbound workgroups.

```html
<html>
<head>
    <title>Wait for Call</title>
    <meta name=IS_Action_SetForeground>
    <meta name=IS_Action_SelectPage>
    <script type="text/javascript" defer>
        scripter.myQueue.objectAddedHandler = ObjectAdded;
        function ObjectAdded(ObjType, ObjId) {

            // we only want to look at call objects (type 2)
            if (2 == ObjType) {
                scripter.callObject.id = ObjId;
                var workgroup = scripter.callObject.getAttribute("EIC_AssignedWorkgroup") + "";
                // adding an empty string guarantees a string
                if ("" == workgroup) {
                    // legacy support
                    workgroup = scripter.callObject.getAttribute("AssignedWorkgroup") + "";
                    // adding an empty string guarantees a string
                }
                // compare case insensitive
                switch (workgroup.toLowerCase()) {
                    //depending on the workgroup name we pop the appropriate page
                    case "sales":
                        window.location.href = "http://server/campaigns/sales";
                        break;
                    case "marketing":
                        window.location.href = "http://server/campaigns/marketing";
                        break;
                    default:
                        // this is not a call from a workgroup that we care about
                        break;
                }
            }
        }
    </script>
</head><body>
    <p>Wait for Call</p>
</body>
</html>
```

Play Digits To a Call

The script below demonstrates how to play digits to a call from within Scripter and a web page.

```javascript
function PlayDigitsToCall(p_CallID, p_DigitsToPlay) {
    var p_Call = scripter.CreateCallObject();
    p_Call.Id = p_CallID;
    p_Call.playDigits(p_DigitsToPlay);
    p_Call = null;
}
```

Preview Campaigns

Preview campaigns allow agents to view contact data prior to actually dialing the call. In addition, preview campaigns dial exactly one call per agent and without any call analysis. So, the call will never be abandoned. To setup a campaign to be a preview campaign, simply select Preview from the Calling Mode drop-down list in the campaign configuration. While developing a script to support these campaigns, the following Scripter functions will be used:

- IS_Event_PreviewDataPop
- IS_Action_PlacePreviewCall
- IS_Action_SkipPreviewCall

Optionally, the IS_Event_NewPreviewCall may be used.

IS_Event_PreviewDataPop

This event signifies that a new contact record is ready to be dialed. The contact data is available and further actions is required to either dial the call (IS_Action_PlacePreviewCall) or skip dialing the contact (IS_Action_SkipPreviewCall).

IS_Event_NewPreviewCall

This event signifies that a preview call is presented to the agent. This is not the data pop. Rather this function indicates that the IS_Action_PlacePreviewCall action was issued. The primary idea behind a preview call is that the agent has the opportunity to learn some details about a contact before the call is placed. Accordingly, when a contact record is received, the script should display information about that record to the agent. This will be done by opening another page that will use IS_Attr commands to display information on the screen.

After viewing the information, an agent would typically click a button to call that contact. The IS_Action_PlacePreviewCall function will place a preview call to the phone number in the sole contact field associated with that campaign. Later, we will discuss dialing alternate phone numbers.

Alternatively, the script might automatically dial a contact without agent intervention. After the information displays for a set time, Dialer would automatically dial the contact. Implement this with the javascript setTimeout(command, timeout_ms) function. It might also be desirable to include an option to skip dialing a record. This would allow the agent to be selective about which records should be dialed. The 'skip call' button is optional.

Preview Process Flowchart

Preview Data Pop Process

Preview Call Data Pops → Goto the Preview Pop Page

Preview Pop Page Loads → Populate Page With Call Record Data

Place Call Button → Place The Preview Call

Skip Call Button → Assign Finish Code → Skip Call Record → Return To Call Waiting Page
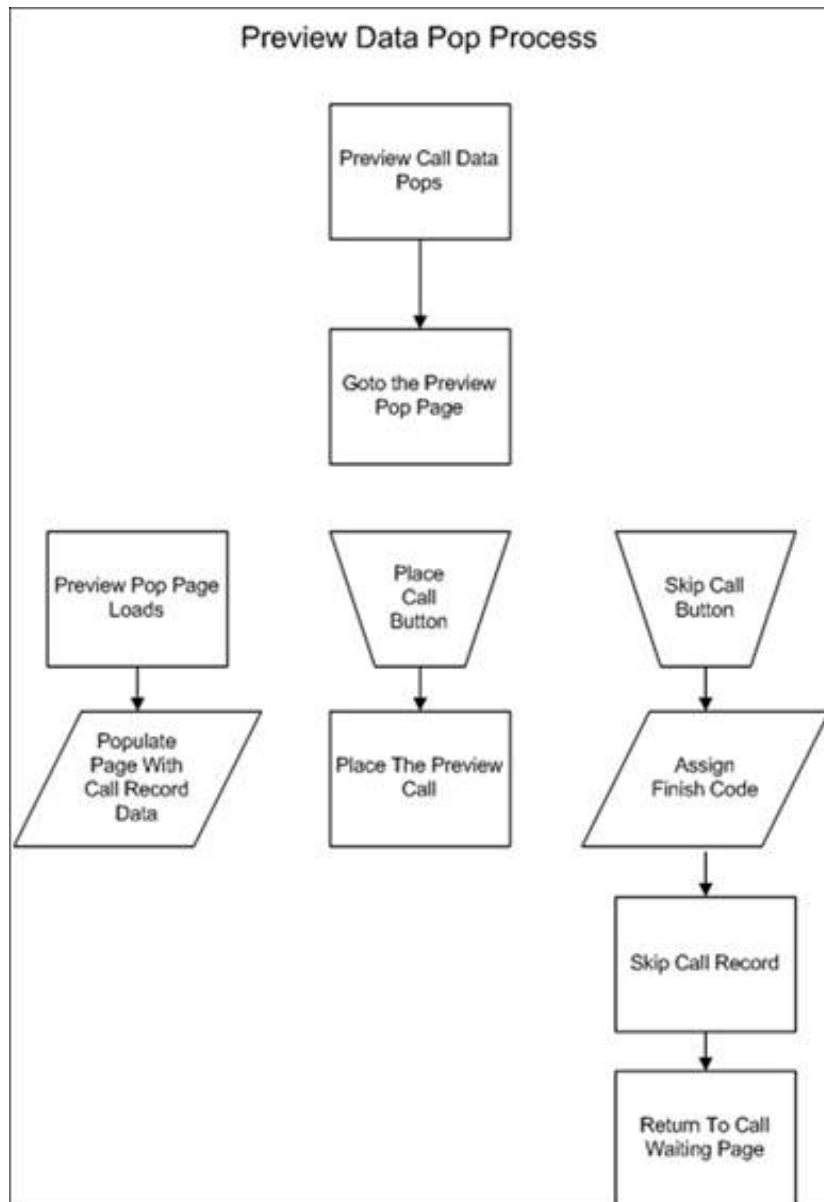
Preview Process Pseudocode

Preview data pop

```
When a preview call record pops:
  Goto the preview pop page
End function
```

Preview call placement

```
When the page loads:
  Populate the page with call record data
End function
When a 'place call' button is pressed:
  Place the preview call
End function
When a 'skip call' button is pressed:
  Skip this call record
  Return to call waiting page
End function
```

Preview Process JavaScript

Preview data pop

```javascript
function IS_Event_PreviewDataPop(p_names, p_values) {
    // display preview data pop
    page location.href = "previewpop.htm";
}
```

Preview call placement

```javascript
window.onload = Init;
function Init() {
    // Populate page with call record data
    tagLName.innertext = IS_Attr_LName;
}
function PlacePreviewCall() {
    // Place the preview call
    IS_Action_PlacePreviewCall.click();
}
function SkipPreviewCall() {
    // Skip this call record
    IS_Action_SkipPreviewCall.click();
    // Return to call waiting page
    location.href = "index.htm";
}
```

Supporting Finishing Agents

In many call centers, the agents that make first contact with a called party are not the agents that ultimately finish the call. Another agent might provide any variety of back-end services from confirming details or getting billing information to performing a satisfaction survey or upselling product. This closer is called a *finishing agent*. While standard agents are the first agents to handle a Dialer call, finishing agents only receive transferred calls. Likewise, scripting support for finishing agents is different from that for standard agents.

Only one script may be linked to a campaign. Consequently, the same script is popped to the standard agents and the finishing agents. It is either up to the agent or to the script developer to display the features appropriate to each agent type. Certainly it would be preferable for the script developer to automate this task.

The following Scripter functions will be used:

- IS_Action_MarkCallForFinishing
- IS_Action_Transfer
- scripter.CallObject.getAttribute

When the time comes for a standard agent to transfer a call to a finishing agent, it is important to mark the call for finishing. Use the IS_Action_MarkCallForFinishing action to accomplish this requirement. Doing so serves two purposes:

- Indicate that the call should only be given to an agent that is logged in as a finishing agent.
- Temporarily write any data associate with the call record, so that it is available to the finishing agent.

When the finishing agent receives that call, the script should detect that this is the back-end of the call and pop an appropriate script. This can be done by setting (and subsequently checking) a custom call attribute.

Related Topics

Transferring Calls

IS_Action_WriteData

IS_Event_QueueObjectRemoved

Transferring Calls

Often, an agent might need to transfer a call. This could be a transfer to a finishing agent, a supervisor, or to another agent. In addition, consider whether the agent should be able to contact the third party prior to transferring. If the call should immediately be transferred to the third party, without notification, we call it a blind transfer. If the agent contacts the third party prior to transferring, we call it a consult transfer.

Blind Transfers

For the blind transfer, the IS_Action_Transfer Scripter function will be used. The IS_Action_Transfer action requires two attributes:

recipient

The extension of phone number of recipient of the transfer.

consult

A flag to designate whether this is a consult transfer action. To perform a blind transfer, simply provide the third-party phone number and execute the transfer.

Consult Transfers

In the case of the consult transfer, these additional Scripter functions will be used:
- IS_Action_Hold
- IS_Action_Pickup
- IS_Action_Disconnect
- scripter.CreateCallObject
- scripter.CallObject.dial
- scripter.CallObject.pickup
- scripter.CallObject.disconnect

When performing a consult transfer, the scripter object will be used to create another call object. This will represent the phone call to the third-party prior to transferring. The script will create the second call and then tear it down when the initial call is actually transferred. The phone number may be a fixed value or may be a field entered by the agent.

Transfer Process Flowcharts
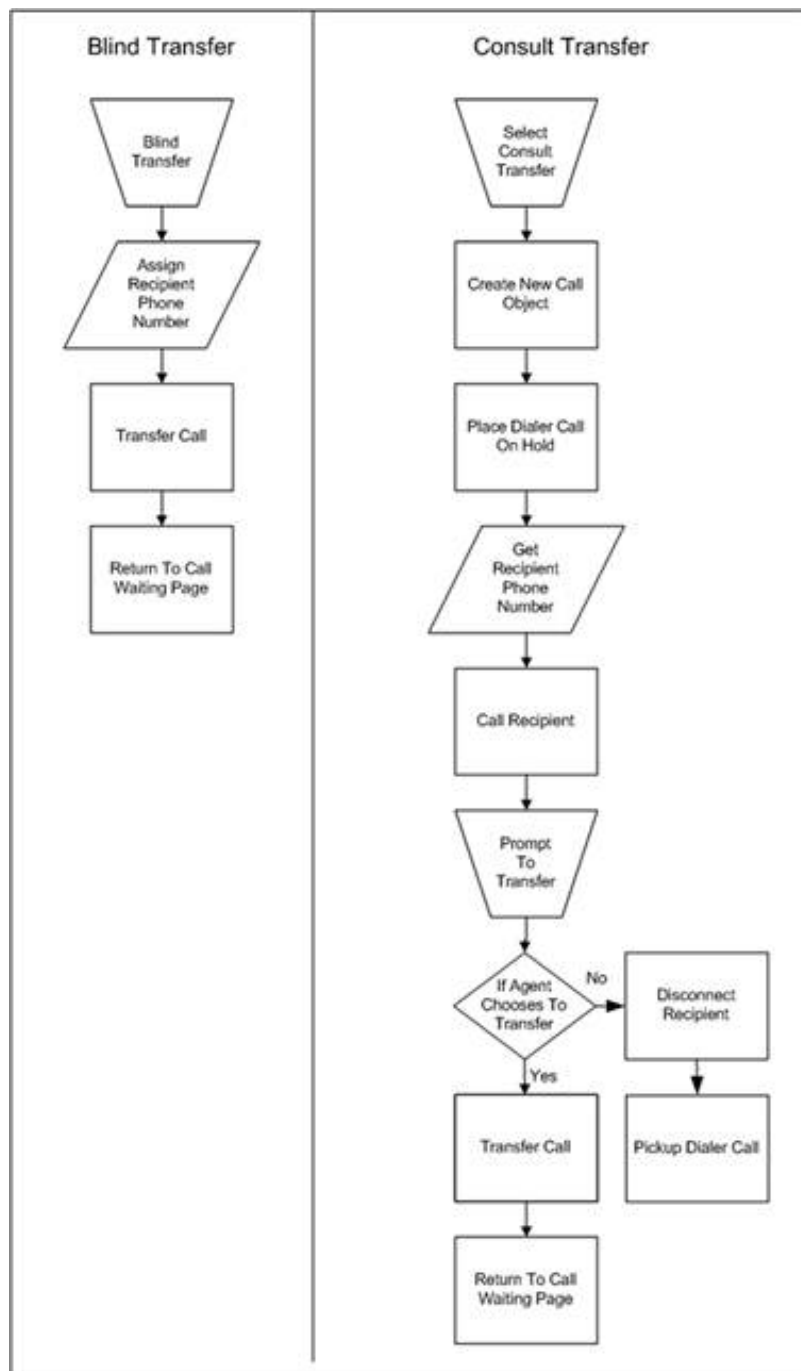
Transfer Process Pseudocode

Blind transfer:

```
When a 'blind transfer' button is pressed:
   Send a break request
End function
```

Consult transfer:

```
When the break is granted:
   Set the agent's status to unavailable
   Goto the designated break page
End function
```

Transfer Process JavaScript

Blind Transfer:

```javascript
function BlindTransfer(p_RecipentPhNum) {

    // Assign the recipient phone number
    IS_Action_Transfer.recipient = p_RecipentPhNum;
    // Designate that this is not a consult transfer
    IS_Action_Transfer.consult = false;

    // Apply the transfer
    IS_Action_Transfer.click();
}
```

Consult transfer:

```javascript
function ConsultTransfer(p_Number) {

    // Create a new scripter call object
    var p_mCallObj = scripter.createCallObject();

    // Prompt the agent to call the recipient
    var iRes1 = confirm("Would you like to call the 3rd party?");
    // user selected OK, so let's call the 3rd party
    if (iRes1) {

        // Dial the recipient
        p_mCallObj.dial(p_Number, false);

        // set up the consult transfer
        IS_Action_Transfer.consult = true;

        // set up the recipient call object to match the person just consulted
        IS_Action_Transfer.recipient = p_mCallObj.id;
        // prompt the agent to transfer
        var iRes2 = confirm("Press OK when you are ready to transfer the call");

        // Transfer call to third party
        if (iRes2) {
            scripter.callObject.id = IS_ATTR_CallID.value;
            // pick up the call, it is probably on hold
            scripter.callObject.pickup();
            // now execute the consult transfer that has been set up
            IS_Action_Transfer.click();
        } else {
            // they did not want to transfer, so lets disconnect the 3rd party
            //call and pick up the original call
            //disconnect 3rd party call
            p_mCallObj.disconnect();
            // pick up original call
            scripter.callObject.id = IS_ATTR_CallID.value;
            scripter.callObject.pickup();
        }
    }
}
```

User Queue Watcher Script

This JavaScript demonstrates how to use the advanced scripting API to set up a queue watcher for the logged in user, and watch for object added and state change events.

```javascript
// Queue Watcher for user queue
scripter.myQueue.objectChangedHandler = CallObjectChanged;
scripter.myQueue.objectRemovedHandler = CallObjectRemoved;
scripter.myQueue.callObjectAddedHandler = CallObjectAdded;
// global call object
var mg_callObj = scripter.createCallObject();
// function called when call object is added to UserQueue
function CallObjectAdded(p_CallObject) {
    mg_callObj = p_CallObject;
}
function ObjectAdded(p_Type, p_ObjId) {
    if (p_Type == 2)
    // only interested in call objects
    {
        mg_callObj.id = p_ObjId;
    }
}
function CallObjectChanged(p_Type, p_ObjId) {
    // will fire whenever object changes state
    if (p_Type == 2) //only interested in call objects
    {
        mg_callObj.id = p_ObjId;
    }
}
function CallObjectRemoved(p_Type, p_ObjId) {
    // will fire when object has been destroyed, which is usually two minutes after disconnect.
    if (p_Type == 2)
    //only interested in call objects
    {
        mg_callObj.id = p_ObjId;
    }
}
```

Workgroup Queue Watcher Script

The JavaScript below demonstrates how to set up a workgroup watcher within Scripter for watching objects being added and changed in a workgroup.

```javascript
function ConnectToQueue(p_QueueName) {
    g_CustomQueue = scripter.createQueue();
    // valid queue types are
    // 3 station queue
    // 9 user queue
    // 10 workgroup queue
    // 15 line queue

    g_CustomQueue.connect(10, p_QueueName);
    g_CustomQueue.callObjectAddedHandler = callObjectAdded;
    g_CustomQueue.objectChangedHandler = callObjectChanged;
    g_CustomQueue.objectRemovedHandler = callObjectRemoved;
    alert("Connected to Queue" + p_QueueName);
}
function callObjectAdded(p_Call) {
    alert(p_Call.id + " has been added to the queue");
}
function callObjectChanged(p_Type, p_ID) {
    //only concerned with call objects
    if (p_Type == 2) {
        alert(p_ID + " has changed state");
    }
}
function callObjectRemoved(p_Type, p_ID) {
    //only concerned with call objects
    if (p_Type == 2) {
        alert(p_ID + "has been removed from queue");
    }
}
```

Frequently Asked Questions

When does my script need to set an agent's status?

When designing scripts, it is helpful to know which statuses Dialer automatically assigns versus statuses that a custom script needs to explicitly assign. Dialer automatically put agents into the following statuses:

Status	Automatically assigned by Dialer when
Awaiting Callback	When an agent-owned scheduled callback or precisely dialed call is being placed for them.
Follow Up	When their current call disconnects before it is dispositioned.
Campaign Call	When the agent is assigned a Dialer call and receives the screen-pop.

A custom script should manually assign a status in the following scenarios:

Status	When to assign in a custom script
'Available'	When the agent completes a call and needs to go back available to receive another call. This is typically done on page load on the 'Waiting for call' page.
On a Break (or whatever non-ACD status a customer chooses)	Whenever a break is granted by Dialer.

What functions or objects are reserved by Interaction Scripter?

Any function or object that begins with the "name" IS_ is reserved by the Interaction Scripter. Scripter currently uses this naming convention with various types of objects:

IS_Attr_

> Attr stands for "attribute". An attribute is a piece of information about an object that travels with an object throughout CIC. An example of an attribute of an object might be the name of the targeted party called. Any Dialer database attribute beginning with IS_Attr_ will automatically become associated with a script object of the same name. If the attribute is first declared in the script, it will go back to the Dialer server during a call complete function, and it can be accessed from a handler. IS_ATTR_ variables are persistent, and are reserved by Interaction Scripter. These values are prepopulated by columns of the same name in a database (less the IS_Attr_ part) under the "value" member for <input> and <meta> elements, and under the "innerText" member for block elements (elements with closing tags like <p></p>).   Any other members that you attach to these elements (i.e. IS_Attr_Foo.bar) are also global. Don't try and attach objects that go out of scope as they will be invalid on the next page (i.e. IS_Attr_Foo.bar = this).

IS_Action_

> Dialer Actions can be performed only on campaign calls. Dialer Actions are also useful in preview mode, when information about a party is pushed to an agent before the agent initiates the call.

IS_Event_

> Events are script callback functions. Events are notification messages from the server, such as new call on a queue. All Dialer Events are functions that you declare, and call when an event occurs.

IS_System_

> System elements retrieve information about an agent, such as the agent's name, or ID, and also retrieve available status messages from the server.

Why is the META tag used?

You may be wondering why some variables are "declared" with a "meta" tag. The meta tag is used for elements that need not be displayed in the HTML page. Since <meta> elements are nonvisual, no special tricks are needed to hide meta elements from the visual part of the document.

One exception is the **IS_CommandToolbar_Visible** meta tag. It allows a script to control whether or not the command toolbar is visible. To display the toolbar, include this meta tag in the head section of a page:

```
<meta name="IS_CommandToolbar_Visible" content="true">
```

To hide the toolbar:

```
<meta name="IS_CommandToolbar_Visible" content="false">
```

How is information sent back to the database?

When declared within the handlers and changed in the Scripter, tagged information is mapped and updated in the database as follows:

All variables named IS_Attr_ are sent to the server for update to the database. Any variables that do not have a corresponding database column are discarded at the time of update.   If the client needs additional information from another database, we recommend that the client query it from an Active Server or Cold Fusion server rather than delaying the primary data pop to the client.

What actually resets and removes script variables after a call has been completed?

Nothing. It is entirely possible to pollute one call with data from the previous. Using only variables from the database, they will be overwritten each time a call is received. Any IS_Attr_ variables that your page defines must be managed and cleared manually.

Is there a way to make variables persistent across all sessions (tabs)?

No. It might be possible to do this with an ActiveX addin control, but no such support is currently built in, nor planned.

Does Scripter support multiple call sessions?

No. Interaction Scripter does not support multiple sessions with a common username or station name. It does not allow multiple calls to be open at the same time.

Can I run a web-based application within Scripter?

Yes. You can run a web-based application (such as Cold Fusion) within Interaction Scripter (on a separate tab) while the agent is running a script for a campaign. Think of these non-campaign pages as regular browser pages with global variables and no security. However, if you allow these pages to browse the web, users might download dangerous content.

What limitations does Scripter have regarding its browsing capabilities?

Only practical limitations apply to the maximum number of sessions. For example, having 100 tabs open makes it quite difficult to find the tab of interest.

Can an application on one tab send information to, or trigger events on another tab?

Cross page functionality is not currently supported or planned for the product, except for IS_Action_Logon that opens an outbound script tab.

What methods change focus between tabs?

Use the "IS_Action_SetForground" method to set focus to the application. A specific page can be brought to the foreground using the "IS_Action_SelectPage" method. And of course, clicking on a tab gives it focus.

Is JavaScript case-sensitive?

Yes. When you define an identifier in JavaScript, the name of the identifier is case-sensitive. For example, if you define:

`<meta name=IS_Action_foo>`

then you must call it using

`IS_Action_foo.click();`

Likewise, if you define:

`<meta name=IS_Action_FoO>`

then you must call it using:

IS_Action_FoO.click();

Does Interaction Scripter support frames?

Interaction Scripter does not directly support frames. Frames can be used, so long as functions in child frame windows call objects and functions in the top-level window. Scripter only recognizes objects and functions in the top-level window. These objects and functions in the top-level window must call down through the object model to the child windows. This requires a working knowledge of the well-documented Internet Explorer Document Object Model, and is entirely the responsibility of the end user.

If frames are used, script developers should give each frame a name using the name attribute of the frame or iframe element. This allows Interaction Scripter to identify events that are specific to the frames, which can result in improved performance.

Does Scripter provide a debugger?

Yes. Interaction Scripter offers a debugging feature that helps developers detect and resolve problems with custom campaign scripts. The debugger is available only when Interaction Scripter is started with a /debug command-line parameter. The syntax is: interactionscripter.exe /debug.

For more information, see [Interaction Scripter Debugger](#)

How do I associate my custom script with a campaign?

Open the campaign in Interaction Administrator. In the scripting frame, select Custom Script URL, then enter the URL of your script. For more information, see "Campaign Name, Workgroup and Script Settings" in the Dialer in Interaction Administrator section of Interaction Dialer Manager Help.

Copyright and Trademark Information

Interaction Dialer and *Interaction Scripter* are registered trademarks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2000-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Messaging Interaction Center and *MIC* are trademarks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2001-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction Director is a registered trademark of Genesys Telecommunications Laboratories, Inc. *e-FAQ Knowledge Manager* and *Interaction Marquee* are trademarks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2002-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction Conference is a trademark of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2004-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction SIP Proxy and *Interaction EasyScripter* are trademarks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2005-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction Gateway is a registered trademark of Genesys Telecommunications Laboratories, Inc. *Interaction Media Server* is a trademark of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2006-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction Desktop is a trademark of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2007-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction Process Automation, Deliberately Innovative, Interaction Feedback, and *Interaction SIP Station* are registered trademarks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2009-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction Analyzer is a registered trademark of Genesys Telecommunications Laboratories, Inc. *Interaction Web Portal* and *IPA* are trademarks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2010-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Spotability is a trademark of Genesys Telecommunications Laboratories, Inc. ©2011-2017. All rights reserved.

Interaction Edge, CaaS Quick Spin, Interactive Intelligence Marketplace, Interaction SIP Bridge, and *Interaction Mobilizer* are registered trademarks of Genesys Telecommunications Laboratories, Inc. *Interactive Intelligence Communications as a Service* and *Interactive Intelligence CaaS* are trademarks or service marks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2012-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction Speech Recognition and *Interaction Quality Manager* are registered trademarks of Genesys Telecommunications Laboratories, Inc. *Bay Bridge Decisions* and *Interaction Script Builder* are trademarks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2013-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interaction Collector is a registered trademark of Genesys Telecommunications Laboratories, Inc. *Interaction Decisions* is a trademark of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2013-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

Interactive Intelligence Bridge Server and *Interaction Connect* are trademarks of Genesys Telecommunications Laboratories, Inc. The foregoing products are ©2014-2017 Genesys Telecommunications Laboratories, Inc. All rights reserved.

The veryPDF product is ©2000-2017 veryPDF, Inc. All rights reserved.

This product includes software licensed under the Common Development and Distribution License (6/24/2009). We hereby agree to indemnify the Initial Developer and every Contributor of the software licensed under the Common Development and Distribution License (6/24/2009) for any liability incurred by the Initial Developer or such Contributor as a result of any such terms we offer. The source code for the included software may be found at http://wpflocalization.codeplex.com.

A database is incorporated in this software which is derived from a database licensed from Hexasoft Development Sdn. Bhd. ("HDSB"). All software and technologies used by HDSB are the properties of HDSB or its software suppliers and are protected by Malaysian and international copyright laws. No warranty is provided that the Databases are free of defects, or fit for a particular purpose. HDSB shall not be liable for any damages suffered by the Licensee or any third party resulting from use of the Databases.

Other brand and/or product names referenced in this document are the trademarks or registered trademarks of their respective companies.

DISCLAIMER

GENESYS TELECOMMUNICATIONS LABORATORIES (GENESYS) HAS NO RESPONSIBILITY UNDER WARRANTY, INDEMNIFICATION OR OTHERWISE, FOR MODIFICATION OR CUSTOMIZATION OF ANY GENESYS SOFTWARE BY GENESYS, CUSTOMER OR ANY THIRD PARTY EVEN IF SUCH CUSTOMIZATION AND/OR MODIFICATION IS DONE USING GENESYS TOOLS, TRAINING OR METHODS DOCUMENTED BY GENESYS.

Genesys Telecommunications Laboratories, Inc.
2001 Junipero Serra Boulevard
Daly City, CA 94014
Telephone/Fax (844) 274-5992
www.genesys.com

Compliance

Please note that it is the sole responsibility of the user of this software to comply with all federal, state, and local laws applicable to the software, the use thereof, and the conduct of the user's business. In no event will Genesys Telecommunications Laboratories, Inc. ("Genesys") be responsible for providing, implementing, configuring, or coding software in a manner that complies with any laws or regulatory requirements that apply to the user's business or industry, including, without limitation, U.S. Federal Trade Commission (FTC) regulations, Federal Communications Commission (FCC) regulations, the Telephone Consumer Protection Act (TCPA) of 1991, and the Health Insurance Portability and Accountability Act (HIPAA) (collectively "Customer Specific Laws"). The user agrees that it will comply with all such Customer Specific Laws and, regardless of anything to the contrary, in no event will Genesys, its affiliates, or related entities be held liable for any claim or action arising from, or related to, the user's failure to comply with any Customer Specific Laws. The above conditions apply regardless of anything to the contrary and your use of Interaction Dialer constitutes your acceptance of the above provisions.

Change log

Date	Changes
11-November-2011	Interaction Scripter's reason code based call completion actions were reworked for Dialer 4.0. The following actions were removed from the documentation because they are now deprecated: IS_Action_AnsweringMachine IS_Action_BeginBreak IS_Action_CallComplete IS_Action_Deleted IS_Action_DisableTransitionDialog IS_Action_EnableTransitionDialog IS_Action_Failure IS_Action_Fax IS_Action_Logoff IS_Action_NoAnswer IS_Action_NoLines IS_Action_PhoneNumberDeleted IS_Action_PhoneNumberSuccess IS_Action_RemoteHangUp IS_Action_Schedule IS_Action_SIT IS_Action_Success IS_Action_Transferred IS_Action_WrongParty
11-November-2011	These actions were replaced by one new action, IS_Action_CallComplete, which should be used to disposition calls. It requires an attribute named *wrapupcode* that contains the wrap up code to be used when dispositioning the call. It supports a Boolean attribute named *abandoned* which indicates whether the call should be classified as abandoned. In addition, IS_Action_CallComplete supports all of the schedule-based attributes that the old IS_Action_Schedule action supported.
11-November-2011	The old actions are deprecated, but they are still functional. Existing customer scripts developed for Scripter 3.0 will not be broken. Old actions will generate debug entries when Scripter is run with the /debug flag. Legacy actions must use appropriate system-defined wrap up codes and ignore the wrapupcode attribute on the element (DIALER-6108).
11-November-2011	IS_Action_FullScreen was removed because the application framework that Scripter Client now runs in does not support full screen mode at this time (DIALER-4205).
11-November-2011	Interaction Dialer no longer supports workflows. Instead, it allows multiple campaigns to run at the same time. For this reason, two workflow-related attributes are no longer supported, even in legacy scripts. The removed attributes are IS_Attr_WorkflowId, which contained the name of the workflow that a campaign object belonged to, and IS_Attr_WorkflowUUID, which contained the id number of the workflow. A third attribute, IS_Attr_CampaignUUID was also deprecated. It contained the campaign ID. It is replaced by a new IS_Attr_CampaignID attribute. In Dialer 4, customers should use these attributes instead of deprecated attributes (DIALER-6214): • IS_Attr_CampaignName is a new attribute which contains the campaign name. • IS_Attr_CampaignID now contains the campaign ID. In earlier releases, it contained the campaign name. Two predictive "NonDialerCallScripting" actions are now used to delay closing a script when the agent has logged out of all the campaigns that the script is associated with (DIALER-6594). • IS_Action_BeginNonDialerCallScripting prevents the agent from being logged out of one campaign and into another while the agent is on a non-Dialer call. Automatic campaign login will be delayed until the agent ends the non-dialer call. • IS_Action_EndNonDialerCallScripting tells Scripter to resume automatic login of an Agent to a new campaign after transitioning was delayed by IS_Action_BeginNonDialerCallScripting. IS_Action_RequestLogoff now accepts an optional *campaigns* attribute, which can be used to request a logout for specific campaigns. If that attribute is not populated, then the action requests a logout for all campaigns. The format of the attribute is a list of campaign names separated by commas.

18-April-2012	The IS_Action_Logon action no longer requires an agent type attribute. Since agent type can only be set for the agent's entire Dialer session and not per campaign, it no longer makes sense to specify it on this action.
28-February-2013	Added scripterNotiferName property. It returns the machine name of the server that the agent is currently logged into.
28-February-2013	Added campaignsChangedHandler callback property, invoked whenever a user is logged into or out of a campaign.
28-February-2013	Added a campaign object with properties that encapsulate the Name, ID and Status of a Dialer campaign.
28-February-2013	Added User.startAvailableCampaignObjectEnum Property to the User object. This property returns an enumeration of accessible queues that current agent has rights to view and modify.
28-February-2013	Added a new element to the Scripter environment called Behaviors. At this time there is only one Predictive Behavior called IS_Bhvr_SuppressToast. Custom scripts can now stream audio files from the ODS server by specifying http or https URI's.
04-November-2014	Updated documentation to reflect changes required in the transition from version 4.0 SU# to Interaction Dialer 2015 R1, such as updates to product version numbers, system requirements, installation procedures, references to Interactive Intelligence Product Information site URLs, and copyright and trademark information.
04-November-2014	Added three new PreviewTimeout Events, which are Predictive Events that are specifically designed for use with Preview campaigns that use a preview countdown timer.
10-February-2015	Added two new methods that can be used to prevent sensitive information from being recorded: • CallObject.pauseSecureRecord - This method can be used to pause recording. • CallObject.resumeSecureRecord - This method resumes recording.
10-February-2015	Added information to the PreviewTimeout Events topic that pertains to the new Manual Calling feature.
10-February-2015	Added clarification that the IS_Action_CallComplete does not disconnect the call.
10-February-2015	Updated the IS_Action_StartReceivingCalls Predictive action topic with information pertaining to the use of the Dialer StartReceivingCalls Per Campaign server parameter.
10-February-2015	Added more details about using the Interaction Scripter Debugger.
05-May-2015	Updated various topics with code examples
03-November-2015	Updated documentation and screen shots to reflect new logo and color scheme.
09-February-2016	The IS_Attr_CampaignGroup Predictive Attribute can now provide the name of the active Campaign Group in the Campaign Sequence that is currently running when using the Advanced Campaign Management feature.
09-February-2016	You can learn more about this new feature in the Advanced Campaign Management for Interaction Dialer - Overview Guide, which is available in the Interaction Dialer section of the CIC Documentation Library.
03-May-2016	The IS_Action_LogonAll Predictive Action is a new action that can globally log an agent into Dialer as well as all currently running campaigns.
09-August-2016	The IS_Action_Exit Standard Action has a new attribute that provides you with ability to exit Dialer as well as Scripter.
01-November-2016	Added IS_Event_ContactDataLoaded and IS_Event_ManualOutboundCallStatus events.
01-November-2016	Added IS_Action_ManualOutboundCall and IS_Action_RequestContactData actions.
08-August-2017	Improved appearance of code listings and topics in this document.
08-August-2017	Updated Copyright and Trademark Information.
08-August-2017	Applied Genesys terminology.

20-February-2018	A new meta tag allows a script to show or hide the command toolbar. See <u>Why is the META tag used?</u>
20-February-2018	Revised Example 2 in <u>IS_Action_Transfer</u> to correct a misspelled method name. The line `onclick="IS_BlindTransferl('101')` is now `onclick="IS_BlindTransfer('101')`.
20-February-2018	Revised <u>IS_Action_Listen</u> to state that CallID is a required attribute. Also updated code samples in that topic.
20-February-2018	Updated the <u>IS_Actions_ClientStatus</u> topic to ensure that only default statuses are listed, and that the case of statuses matches CIC default status names in PureConnect .
15-May-2018	**Interaction Connect now supports custom outbound scripts**. Starting with PureConnect 2018 R3, call center agents can use **Interaction Connect** to process outbound Dialer calls with custom script screen pops. Previously, Interaction Connect supported base scripts only. Agents were required to use **Interaction Scripter .NET** client for campaigns with custom scripts. To use Interaction Connect instead, custom scripts must conform with new programming requirements.
15-May-2018	Interaction Connect processes JavaScript asynchronously, breaking tasks into threads that execute independently. Conversely, Interaction Scripter .NET is synchronous. It waits for each script statement to finish before moving to the next. For this reason, Interaction Connect and Scripter .NET have divergent programming requirements. • Custom scripts must be written for one client or the other. • The same custom script cannot be used in both Scripter .NET and Interaction Connect. • Legacy scripts written for Interaction Scripter .NET work as before in that client application. • To be compatible with Interaction Connect, scripts must implement callback functions that indicate when asynchronous operations complete. • Scripts written for Scripter .NET should not implement callbacks designed for Connect. For more information, see <u>Writing custom scripts for Interaction Connect or Scripter .NET</u>, <u>Sample Interaction Connect scripts</u>, and <u>Sample ICWS Dialer Web Application</u>.
15-May-2018	<u>IS_Action_CallComplete</u> has a new Boolean parameter named MakeFollowUpCall. It indicates whether the user should be put into "Additional Follow Up status", in support of a feature that allows an agent to dial additional calls while in that status.
15-May-2018	The <u>IS_ActioStartReceivingCalls</u> action now accepts an optional new campaign attribute, containing a comma-separated list of campaign names for the Dialer agent to receive calls from.
15-May-2018	<u>IS_Event_ManualOutboundCallStatus</u> has updated arguments. `Status` now returns a code number corresponding to `StatusName`, a new attribute that returns the status string.
15-May-2018	The <u>ConferenceObject.subObjectChangedHandler</u> callback now has two parameters instead of three.
15-May-2018	Documented a new event: <u>IS_Event_PreviewCallSkipped</u> is raised when a preview call is successfully skipped.
15-May-2018	Added a new standard action: <u>IS_Action_CompleteConsult</u>. This action ends a consult call in scripts for Interaction Connect only.
15-May-2018	Updated <u>IS_Action_Transfer</u> to add an "audience" parameter. This new parameter is used only with scripts for Interaction Connect.
15-May-2018	Documented <u>capitalization conventions</u> to use when writing scripts.
15-May-2018	A new action, <u>IS_Action_QueryContactList</u> queries a specified contact list for records.
15-May-2018	Two conference object callbacks were added for use in scripts for Interaction Connect. <u>ConferenceObject.conferenceObjectInitializedHandler</u> is invoked when the conference object has initialized. <u>ConferenceObject.conferenceStartedHandler</u> is invoked when the conference call has started.
15-May-2018	A new <u>ChatObject.messages</u> property returns an array of messages about a chat interaction after the chat object has been initialized.
15-May-2018	A new <u>ChatObject.sendChatMessage</u> method sends a chat message to the remote party on behalf of the current user.

15-May-2018	A new ChatObject.messages property of a chatObject returns an array of messages of that chat interaction after the chatObject had been initialized.
15-May-2018	The ChatObject.SubObjectChangeHandler can now be an assigned callback that gets alerted with new messages when the chat Interaction receives new messages.
15-May-2018	A new CallObject.currentDialerCallId method returns the Id of the current active dialer interaction. This can be used to set a CallObject's id to initialize it with the current active dialer call.
15-May-2018	A new dialer.sendCustomHandlerNotification method initiates custom handlers through the custom notification initiator.
15-May-2018	A new dialer.subscribeToCustomHandlerNotification method allows a custom script to asynchronously subscribe and listen to a Send custom notification step in a custom handler.
15-May-2018	Fixed a bug that could prevent an agent in Available, Forward status from entering a phone number or optional notes while using a custom script.
15-May-2018	Added a note to IS_Action_PlayWav. For custom scripts in Scripter Connect, the wav file specified for the IS_Action_PlayWav action must be located in the Resource Path directory on the CIC server (I3\IC\Resources by default).
15-May-2018	Interaction Connect now allows agents executing a base script to change their user status when they are not involved in a Dialer interaction. Agents can change their status before logging into a campaign, or while on break.
15-May-2018	A new action, IS_Action_PlaceChat initializes a chat between the current user and another user.
15-May-2018	A new action, IS_Action_PlaceChat initializes a chat between the current user and another user. When the chat is initialized, a new IS_Event_ChatInitialized event is emitted. Your scripts should listen for this event to ensure that they do not proceed until the chat is fully initialized.
15-May-2018	The ConferenceObject.create method now has 1 or 2 input parameters, depending upon whether the script will run in Scripter .NET or Interaction Connect. To create a conference in an Interaction Connect script, two interaction id's must be passed to this method. Scripter .NET requires a single interaction ID to be passed.
15-May-2018	The Queue object for custom scripts in Connect now deviates from Scripter.NET in a couple of ways. • The following Queue object properties are compatible with Scripter .NET, but not Interaction Connect: Queue.errorHandler Queue.lastError Queue.lastErrorId • The Queue.connect method can be used asynchronously in Connect. It now accepts a callback argument that takes no parameters. In addition, Queue.connect no longer supports Line Queue as a Type parameter. • Queue.errorhandler, Queue.lastError, and Queue.lastErrorId should not be used in scripts for Connect, but may be used as before in scripts for Scripter .NET. • The Queue.startCallObjectsEnum, Queue.startChatObjectsEnum, Queue.startConferenceObjectsEnum and Queue.startObjectIdsEnum properties now accept an optional callback (for use with Connect scripts only) whose single parameter contains the result.
15-May-2018	Similarly, the User object is programmed differently for Connect scripts: • The User.errorHandler, User.lastError, and User.lastErrorId properties are compatible with Scripter .NET as before, but these properties should not be used in scripts for Interaction Connect. • The User.startAccessibleQueuesEnum and User.startViewableWorkgroupsEnum properties now accept an optional callback (for use with Connect scripts only) whose single parameter contains the result.
15-May-2018	Noted in the Interaction Scripter Actions topic that a few actions should be used with care, since they potentially affect the performance of a script or server. Actions marked as having scale impact include: • IS_Action_QueryContactList—because an inefficient or overly broad query could affect the performance of a database server. • Dialer.sendCustomHanderlNotification Method—starting a very complex, long-running handler could affect the performance of a PureConnect server.

21-August-2018	Scripter 2018 R3 documentation incorrectly described a Boolean parameter of the IS_Action_CallComplete action. The name of the parameter is makefollowupcall, not makeadditionalfollowupcall. Documentation was corrected.
13-November-2018	Updated Example 2 on the IS_Action_Record page for use with snippet recordings by including "IS_Action_Record.callid = IS_Attr_CallId.value;" in the Record() function.
13-November-2018	The Sample Interaction Connect scripts now include a new multi-page custom script that demonstrates some of the key actions and events you can use when writing a custom script in Interaction Scripter.
13-November-2018	You must now specify a callback function when using IS_Action_PlaceChat, in order to ensure that script execution doesn't proceed until the chat is fully initialized. The second sample script has been updated to reflect this.
05-May-2019	Added How do I associate my custom script with a campaign? to the FAQs.
03-June-2019	Added notes indicating that the dialer.sendCustomHandlerNotification Method and the dialer.subscribeToCustomHandlerNotification Method can only be used with Interaction Connect and not with Scripter .NET.
05-February-2020	Removed incorrect StateString values from CallObject.stateChangeHandler Callback Property.