SALT

Speech Application Language Tags (SALT)
1.0 Specification

The information contained herein is not a license, either expressly or impliedly, to any intellectual property owned or controlled by any of the authors or developers of this specification. The information contained herein is provided on an "AS IS" basis and to the maximum extent permitted by applicable law, this information is provided AS IS AND WITH ALL FAULTS, and the authors and developers of this specification hereby disclaim all other warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties, duties or conditions of merchantability, of fitness for a particular purpose, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence, all with regard to the information and their contribution thereto. ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO THE INFORMATION.

IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF THIS SPECIFICATION BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, PUNITIVE OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO THE SPECIFICATION INFORMATION, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

For more information, please contact the SALT Forum at mailto:info@saltforum.org or visit http://www.saltforum.org.
Table of Contents

1 Introduction .......................................................................................................................... 5
  1.1 Overview .......................................................................................................................... 5
  1.2 Scenarios ........................................................................................................................ 5
  1.3 Design principles ............................................................................................................ 7
    1.3.2 Dynamic manipulation of SALT elements ................................................................. 7
    1.3.3 Events and error handling ....................................................................................... 7
    1.3.4 Management of external resources .......................................................................... 8
  1.4 Document structure ........................................................................................................ 8
  1.5 Terms and definitions .................................................................................................... 9

2 SALT speech interface ........................................................................................................ 11
  2.1 Speech output: <prompt> ............................................................................................... 11
    2.1.1 prompt content ........................................................................................................ 12
    2.1.2 prompt attributes and properties ........................................................................... 14
    2.1.3 prompt methods ...................................................................................................... 14
    2.1.4 prompt events ........................................................................................................ 15
    2.1.5 PromptQueue object .............................................................................................. 17
  2.2 Speech input: <listen> .................................................................................................... 26
    2.2.1 listen content .......................................................................................................... 26
    2.2.2 listen attributes and properties ............................................................................. 30
    2.2.3 listen methods ........................................................................................................ 31
    2.2.4 listen events ........................................................................................................... 33
    2.2.5 Interaction with DTMF .......................................................................................... 36
    2.2.6 Recognition mode .................................................................................................. 36
    2.2.7 Events which stop listen execution ....................................................................... 42
    2.2.8 Recording with listen ........................................................................................... 42
    2.2.9 Advanced speech recognition technology ............................................................ 48
  2.3 DTMF input: <dtmf> ........................................................................................................ 48
    2.3.1 dtmf content ........................................................................................................... 48
    2.3.2 dtmf attributes and properties ............................................................................. 49
    2.3.3 dtmf methods ........................................................................................................ 50
    2.3.4 dtmf events ............................................................................................................. 51
    2.3.5 DTMF event timeline ............................................................................................ 53
    2.3.6 Using listen and dtmf simultaneously .................................................................. 54
    2.3.7 Events which stop dtmf execution ....................................................................... 56
  2.4 Platform messaging: <smex> ........................................................................................ 57
    2.4.1 smex content .......................................................................................................... 57
    2.4.2 smex attributes and properties ............................................................................. 58
    2.4.3 smex events ............................................................................................................. 58
    2.4.4 Using smex for telephony call control ................................................................ 59
  2.5 Logging ............................................................................................................................. 59
    2.5.1 Overview ................................................................................................................ 59
    2.5.2 Format ..................................................................................................................... 59
    2.5.3 Requirements ......................................................................................................... 59
  2.6 SALT illustrative examples .............................................................................................. 60
    2.6.1 Controlling dialog flow ......................................................................................... 60
    2.6.2 Prompt examples ..................................................................................................... 64
    2.6.3 Using SMIL ............................................................................................................. 66
    2.6.4 Wireless Phone (WML) example .......................................................................... 67
1 Introduction

Speech Application Language Tags (SALT) 1.0 is an extension of HTML and other markup languages (cHTML, XHTML, WML, etc.) which adds a speech and telephony interface to web applications and services, for both voice only (e.g. telephone) and multimodal browsers.

This section introduces SALT and outlines the typical application scenarios in which it will be used, the principles which underlie its design, and resources related to the specification.

1.1 Overview

SALT is a small set of XML elements, with associated attributes and DOM object properties, events and methods, which may be used in conjunction with a source markup document to apply a speech interface to the source page. The SALT formalism and semantics are independent of the nature of the source document, so SALT can be used equally effectively within HTML and all its flavors, or with WML, or with any other SGML-derived markup.

The main top-level elements of SALT are:

- `<prompt ...>` for speech synthesis configuration and prompt playing
- `<listen ...>` for speech recognizer configuration, recognition execution and post-processing, and recording
- `<dtmf ...>` for configuration and control of DTMF collection
- `<smex ...>` for general purpose communication with platform components.

The input elements `listen` and `dtmf` also contain grammars and binding controls:

- `<grammar ...>` for specifying input grammar resources
- `<bind ...>` for processing recognition results.

`listen` also contains the facility to record audio input:

- `<record ...>` for recording audio input

`smex` also contains the binding mechanism `bind` to process messages.

All four top-level elements contain the platform configuration element `<param ...>`.

A `PromptQueue` object and `LogMessage` function are also available. For control of telephony functionality, an optional `call control` object is defined (Part 3) and a set of predefined messages for linking SALT with other call control models is also possible (see section 2.4.4).

There are several advantages to using SALT with a mature display language such as HTML. Most notably (i) the event and scripting models supported by visual browsers can be used by SALT applications to implement dialog flow and other forms of interaction processing without the need for extra markup, and (ii) the addition of speech capabilities to the visual page provides a simple and intuitive means of creating multimodal applications. In this way, SALT is a lightweight specification which adds a powerful speech interface to web pages, while maintaining and leveraging all the advantages of the web application model.

1.2 Scenarios

Two major scenarios for the use of SALT are outlined below, with simple code samples. For a fuller description of the elements used in these examples, please see the detailed definitions later in the document.

Multimodal

For multimodal applications, SALT can be added to a visual page to support speech input and/or output. This is a way to speech-enable individual HTML controls for ‘push-to-talk’ form-filling scenarios, or to add more complex mixed initiative capabilities if necessary.

A SALT recognition may be started by a browser event such as clicking on a button, for example, which activates a grammar relevant to an adjacent input field, and binds the recognition result into that field:
Voice-only and telephony

For applications without a visual display, SALT manages the interactional flow of the dialog and the extent of user initiative by using the HTML eventing and scripting model. In this way, the full programmatic control of client-side (or server-side) code is available to application authors for the management of prompt playing and grammar activation. (Implementations of SALT are expected to provide scriptlets which will make easier many common dialog processing tasks, e.g. generic forms of the RunAsk script below or the RunSpeech script illustrated in section 2.6.1.2).

A simple system-initiative dialog might be authored in the following way, for example, where the RunAsk() function activates prompts and recognitions until the values of the input fields are filled:

```html

<body onload="RunAsk()">
  <form id="travelForm">
    <input name="txtBoxOriginCity" type="text" />
    <input name="txtBoxDestCity" type="text" />
  </form>

  <!-- Speech Application Language Tags -->
  <salt:prompt id="askOriginCity"> Where would you like to leave from? </salt:prompt>
  <salt:prompt id="askDestCity"> Where would you like to go to? </salt:prompt>

  <salt:listen id="recoOriginCity" onreco="procOriginCity()">
    <salt:grammar src="city.xml" />
  </salt:listen>

  <salt:listen id="recoDestCity" onreco="procDestCity()">
    <salt:grammar src="city.xml" />
  </salt:listen>

  <!-- scripted dialog flow -->
  <script>
    function RunAsk() {
      if (travelForm.txtBoxOriginCity.value=="") {
        askOriginCity.Start();
        recoOriginCity.Start();
      } else if (travelForm.txtBoxDestCity.value=="") {
        askDestCity.Start();
        recoDestCity.Start();
      }
    }

    function procOriginCity() {
      travelForm.txtBoxOriginCity.value = recoOriginCity.text;
      RunAsk();
    }

    function procDestCity() {
      travelForm.txtBoxDestCity.value = recoDestCity.text;
      travelForm.submit();
    }
  </script>
```

© SA LT Forum founding companies page 6
1.3 Design principles

SALT is designed to be a lightweight markup layer which adds the power of a speech interface to existing markup languages. As such it can remain independent (i) of the high-level page in which it is contained (e.g. HTML); (ii) of the low-level formats which it uses to refer to linguistic resources, e.g. the text-to-speech and grammar formats; and (iii) of the individual properties of the recognition and speech synthesis platforms used by a SALT interpreter. In order to promote interoperability of SALT applications, the use of standard formats for external resources is encouraged wherever possible.

SALT elements are not intended to have a default visual representation on the browser, since for multimodal applications it is assumed that SALT authors will signal the speech enablement of the various components of the page by using application-specific graphical mechanisms in the source page.

1.3.1.1 Modes of execution

Since SALT uses the browser environment of the page in which it is hosted to implement its execution model, the level of programmatic access afforded to the DOM interfaces of SALT elements will differ according to the capabilities of those environments. This notion comes most clearly into perspective when browsers with and without event and scripting capabilities are considered. These classes of browser are broadly labeled 'uplevel' and 'downlevel' respectively, and one can think of SALT as running in a different 'mode' in each class of browser: object mode and declarative mode.

Object mode, where the full interface of each SALT element is exposed in the host environment to programmatic access by application code, is available for uplevel browsers such as those supporting HTML events and scripting modules. Object mode offers SALT authors a finer control over element manipulation, since the capabilities of the browser are greater. (For the most part this specification provides illustrations of the SALT objects in object mode. These illustrations typically assume support of the XHTML Scripting and Intrinsic Event Modules, as defined in the W3C XHTML Recommendation at http://www.w3.org/TR/xhtml1.)

Declarative mode, where a more limited interface of each SALT element is directly exposed, but for which the key functionality is still accessible declaratively, is available in downlevel browsers, such as those not supporting event and scripting modules. Such browsers are likely to be smaller devices, without sufficient processing power to support a scripting host, or more powerful classes of device for which full scripting support is not required or desired. In declarative mode, manipulation of the DOM object of SALT elements is typically limited to attribute specification and simple method calling from other elements. As will be seen, such manipulation can be performed through bind statements in the SALT messaging or input modules, for example, or by other browser means if supported (e.g. the declarative multimedia synchronization and coordination mechanisms in SMIL 2.0, as described in 2.8.3).

1.3.2 Dynamic manipulation of SALT elements

In object mode, client-side scripts are able to access the elements of the SALT DOM. For this reason, it is important that SALT implementations address the dynamic manipulation of SALT elements. For example, client-side script may be used to change the value of an event handler:

```html
<salt:listen id="listen1" onreco="regularListenFunction">
  ...
</salt:listen>

<script><![CDATA[
  listen1.onreco="specialListenFunction";
]]></script>
```

This is a well-known execution model with HTML and many other markup/script models. SALT implementations must address the probability that advanced dialog authors may dynamically reconfigure the objects of SALT just before a call to execute them.

1.3.3 Events and error handling

Each of the SALT elements and objects specified in this document defines a set of events associated with the functionality of the element. For example, the onreco event is fired on a listen element when the speech recognition
engine successfully completes the recognition process. The asynchronous nature of eventing in this environment means that applications will typically follow an event-driven programming model. A single textbox, for example, could be updated by values at any time from speech or GUI events. Dialog flow can be authored by triggering selection scripts for dialog turns on the basis of such events.

It should be noted that those properties of SALT objects which are updated by events (for example, the status property on many objects) are considered meaningful for evaluation only in the handler of the relevant event which sets them. For instance, an application should examine the recosresult property of a listen object in the onreco or onnoreco event handlers, and of the bookmark property of a prompt object in the onbookmark event handler.

Each SALT object specifies an onerror event, which when fired signifies a serious or fatal platform exception. The exception updates the element with an associated code in the status property that allows the application developer to decide what the best course of action is for the platform error that is thrown.

1.3.3.1 Event models and notation
Much work has been done and is ongoing in the web community on event models across web environments (e.g. HTML intrinsic events (http://www.w3.org/TR/html4/) and browser derivatives thereof; DOM Level 2 and 3 events (http://www.w3.org/DOM/); XML Events (http://www.w3.org/TR/xml-events/); etc.). Current web pages therefore reflect a diversity of event models. (For an illustration of the HTML event model implemented in Microsoft Internet Explorer and comparison with the W3C DOM Level 2 event model, see section 2.8.2.2.1.3.) Given that SALT does not define an event model itself and there is not yet a single event model which is standard across browsers and profiles, a number of syntaxes are possible for the events which SALT defines. Whichever is used will depend on the event model supported by the profile in which SALT is used.

In the sections that define SALT events, therefore, only the event name is provided. The syntax of using the event is exemplified for a number of common models in the table below, and this may be used as a reference for the use of any SALT event in that environment.

This table applies the syntax of these models to an example event, the onbookmark event of the prompt object, where handler is the name of the function called when the event is thrown, and promptId is the identifier of the prompt object which holds the event:

<table>
<thead>
<tr>
<th>Syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inline HTML</strong></td>
</tr>
<tr>
<td>&lt;prompt onbookmark=&quot;handler&quot; …&gt;</td>
</tr>
<tr>
<td><strong>XML Events</strong></td>
</tr>
</tbody>
</table>
| <ev:listener ev:name="promptId"  
  ev:event="onbookmark"  
  ev:handler="handler" … />                       |
| **Programmatic assignment**                             |
| ECMA Script: Object.onbookmark = handler;             |
| VB Script: Object.onbookmark = GetRef("handler");     |

1.3.4 Management of external resources
SALT applications may require resources held in external documents such as audio files, grammars, script libraries, etc. These resources can be large and/or time-consuming to access and load, and applications often need fine level control over downloading and caching policies. Mechanisms for such management are typically available within the host environments in which SALT is expected to be used, for example the HTTP 1.1 mechanisms in HTML. Hence SALT does not itself include any resource management capabilities (with the exception of the prefetch attribute on the prompt element).

1.4 Document structure
The rest of this specification is structured as follows.

Part 2 describes the SALT speech interface. Sections 2.1 to 2.4 describe the core elements of the SALT markup: prompt, listen, dtmf and smex. Each section details the syntax and semantics of the SALT element, including default behavior, and outlines the element and its associated attributes, properties, methods and events. Chapter 2.5 describes the logging function for the recording of platform events and data. Chapter 2.6 contains a number of examples illustrating
the use of SALT to accomplish a variety of tasks. Chapter 2.7 holds the SALT DTD. Chapter 2.8 introduces modularization and profiling issues and outlines SALT/HTML profiles for multimodal and voice-only scenarios.

Part 3 describes the optional CallControl object that may be available in telephony profiles for the control of telephony functionality, and shows some illustrative examples of how it may be used.

Part 4 specifies conformance criteria for different classes of SALT browser.

1.5 Terms and definitions
Throughout this document, the uses of the words 'must', 'should' and 'may' with respect to requirements on SALT browser behavior are to be interpreted as "MUST" (REQUIRED), "SHOULD" (RECOMMENDED) and "MAY" (OPTIONAL), respectively, as defined in IETF RFC 2119 (http://www.ietf.org/rfc/rfc2119.txt). The conformance section in part 4 defines overall conformance requirements for SALT browsers.

Here are the definitions of some terms used within the specification.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCXML</td>
<td>Call Control eXtensible Markup Language. A markup language for specifying telephony call control applications. Developed by the Voice Browser Working Group at W3C, the initial Working Draft is at <a href="http://www.w3.org/TR/ccxml/">http://www.w3.org/TR/ccxml/</a>.</td>
</tr>
<tr>
<td>CFG</td>
<td>Context-free grammar, such as W3C SRGS (see below).</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model, a standard interface to the contents of a web page, as described at <a href="http://www.w3.org/DOM/">http://www.w3.org/DOM/</a>.</td>
</tr>
<tr>
<td>DTMF</td>
<td>Dual Tone Multi-Frequency. Telephone touch-tone or push-button dialing.</td>
</tr>
<tr>
<td>Downlevel browser</td>
<td>A browser which does not support full eventing and scripting capabilities. This kind of SALT browser will support the declarative aspects of a given element (i.e. rendering of the core element and attributes), but will not expose all the DOM object properties, methods and events for direct manipulation by the application. Downlevel browsers will typically be found on clients with limited processing capabilities.</td>
</tr>
<tr>
<td>Event bubbling / Event propagation</td>
<td>This is the idea that an event can affect one object and a set of related objects. Any of the potentially affected objects can block the event or substitute a different one (upward event propagation). The event is broadcast from the node at which it originates to every parent node.</td>
</tr>
<tr>
<td>JCP</td>
<td>Java Call Processing API. See <a href="http://java.sun.com">http://java.sun.com</a></td>
</tr>
<tr>
<td>JTAI</td>
<td>Java Telephony API. See <a href="http://java.sun.com">http://java.sun.com</a></td>
</tr>
<tr>
<td>Mixed Initiative</td>
<td>A form of dialog interaction model, whereby the user is permitted to share the dialog initiative with the system, e.g. by providing more answers than requested by a prompt, or by switching task when not prompted to do so (see also System Initiative.)</td>
</tr>
<tr>
<td>Multimodal</td>
<td>Describing applications or interactions where more than a single mode of input or output is available to the end-user. For SALT this is used in the case where a speech interface is available in addition to a visual interface.</td>
</tr>
<tr>
<td>N-Gram</td>
<td>A stochastic language model, such as W3C Stochastic Language Models (N-Gram) Specification (<a href="http://www.w3.org/TR/ngram-spec/">http://www.w3.org/TR/ngram-spec/</a>).</td>
</tr>
<tr>
<td>NLSML</td>
<td>Natural Language Semantic Markup Language. W3C specification for representing the meaning of a natural language utterance and associated information. At the time of writing, this specification is at early Working Draft status. The latest version may be found at <a href="http://www.w3.org/TR/nl-spec/">http://www.w3.org/TR/nl-spec/</a>.</td>
</tr>
<tr>
<td>SGML</td>
<td>Standard Generalized Markup Language, a formalism for languages which structure document content. SGML resources may be found at <a href="http://www.w3.org/MarkUp/SGML/">http://www.w3.org/MarkUp/SGML/</a>.</td>
</tr>
<tr>
<td>SRGS</td>
<td>Speech Recognition Grammar Specification. W3C specification for representing speech recognition grammars. The latest version may be found at <a href="http://www.w3.org/TR/speech-grammar/">http://www.w3.org/TR/speech-grammar/</a>.</td>
</tr>
<tr>
<td>SMIL</td>
<td>Synchronized Multimedia Integration Language. A W3C Recommendation, SMIL 2.0</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Speech Application Language Tags (SALT) 1.0</td>
<td>(pronounced “smile”) enables simple authoring of interactive audiovisual applications. See <a href="http://www.w3.org/TR/smil20">http://www.w3.org/TR/smil20</a>.</td>
</tr>
<tr>
<td>SSML</td>
<td>Speech Synthesis Markup Language. W3C specification for controlling the output of a prompt engine. The latest version may be found at <a href="http://www.w3.org/TR/speech-synthesis">http://www.w3.org/TR/speech-synthesis</a>.</td>
</tr>
<tr>
<td>System Initiative</td>
<td>A form of dialog interaction model, whereby the system holds the initiative, and typically drives the dialog with simple questions to which only a single answer is possible. (see also Mixed Initiative.)</td>
</tr>
<tr>
<td>TTS</td>
<td>Text-To-Speech: the synthesis of speech output on the basis of textual input.</td>
</tr>
<tr>
<td>Uplevel browser</td>
<td>A browser which supports full event and scripting capabilities. This kind of SALT browser will support programmatic manipulation of the attributes, properties, methods and events of every given SALT element. Uplevel browsers will typically be found on ‘rich’ clients with full processing capabilities.</td>
</tr>
<tr>
<td>Voice-only</td>
<td>Describing applications or interactions where the speech modality is the only interface available to the end-user, such as typical telephony scenarios.</td>
</tr>
<tr>
<td>WML</td>
<td>Wireless Markup Language, a language developed by the Wireless Application Protocol Forum (<a href="http://www.wapforum.org">http://www.wapforum.org</a>) for applications on wireless devices. Some WML browsers also support WMLScript, a compact scripting language. Specifications may be found at: <a href="http://www.wapforum.org/what/technical.htm">http://www.wapforum.org/what/technical.htm</a></td>
</tr>
<tr>
<td>XHTML</td>
<td>eXtensible HyperText Markup Language. W3C Recommendation which reformulates HTML as an XML application, defined at <a href="http://www.w3.org/TR/xhtml1">http://www.w3.org/TR/xhtml1</a></td>
</tr>
<tr>
<td>XML Name</td>
<td>Syntactic construct for tokens in XML documents, essentially “a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops” (from <a href="http://www.w3.org/TR/REC-xml#NT-Name">http://www.w3.org/TR/REC-xml#NT-Name</a>).</td>
</tr>
<tr>
<td>XPath</td>
<td>XML Path language, a W3C Recommendation for addressing parts of an XML document. See <a href="http://www.w3.org/TR/xpath">http://www.w3.org/TR/xpath</a>.</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transformations, a W3C Recommendation for transforming XML documents. See <a href="http://www.w3.org/TR/xslt">http://www.w3.org/TR/xslt</a>.</td>
</tr>
</tbody>
</table>
2 SALT speech interface

2.1 Speech output: <prompt>
The prompt element is used to specify the content of audio output. The content of prompts may be one or more of the following:

- inline or referenced text, which may be marked up with prosodic or other speech output information;
- variable values retrieved at render time from the containing document;
- links to audio files.

Prompts can be specified and played individually, and, in more complex applications, they may be managed through a model of prompt queuing. In this model, prompts are queued and played in conceptual subqueues and, in those browsers which support the PromptQueue module, subqueue manipulation is available through the PromptQueue object (see section 2.1.5). This model is outlined below.

Prompt queuing model
Before defining prompts and the PromptQueue in detail, it may be useful to explain the conceptual model behind prompt queuing and playback. There are effectively three units of prompt management:

- prompt object (or element)
- prompt subqueue
- PromptQueue object

The prompt object is the smallest unit of manipulation. As defined in 2.1.1, it holds content for playback.

The prompt subqueue is a conceptual sequence of one or more prompt objects. A prompt is added to a subqueue when the prompt is queued. The subqueue is "closed" (that is, it becomes a fixed unit to which no more prompts may be added) by a call to start audio playback. In downlevel browsers, queuing and starting playback typically happen sequentially (activation is mapped to prompt start behavior, as in 2.1.3.2), so the prompt subqueue in a downlevel browser will consist of only a single prompt. In uplevel browsers, individual Queue and Start methods are available to a prompt, so a prompt subqueue may consist of multiple prompts. For example, an application can call the following sequence of methods:

```javascript
prompt1.Queue();
prompt2.Queue();
prompt3.Start();
prompt4.Queue();
prompt5.Start();
```

which builds two subqueues. The first subqueue holds prompt1, prompt2 and prompt3; the second subqueue holds prompt4 and prompt5. Similarly, the following example:

```javascript
prompt1.Start();
prompt2.Start();
prompt3.Start();
```

builds 3 subqueues, holding prompt1, prompt2, and prompt3, respectively. The 3 subqueues will be played sequentially, each subqueue starting immediately after the previous one completes (without the need for subsequent calls to begin playback).

The PromptQueue object is supported in uplevel browsers which implement the PromptQueue module. As defined in section 2.1.5, the PromptQueue object may be thought of as the container of the prompt subqueues. The PromptQueue object provides a means of controlling the subqueue of prompts currently under playback. In relevant HTML profiles the PromptQueue is a child of the window object.

The behavior of prompts, subqueues and the PromptQueue object is explained in greater detail below.
2.1.1 prompt content

The prompt element contains the resources for system output. The content of prompts may be one or more of the following:

- inline or referenced text, which may be marked up with prosodic or other speech output information;
- variable values retrieved at render time from the containing document;
- links to audio files.

It also permits platform-specific configuration using the param element.

2.1.1.1 Text and inline TTS markup

Simple prompts need specify only the text required for output, e.g.:

```xml
<prompt id="Welcome">
  Thank you for calling the weather report.
</prompt>
```

SALT also allows any format of speech synthesis markup language to be used inside the prompt element.

To enable interoperability of SALT applications, SALT browsers must support the W3C Recommendation for Speech Synthesis Markup Language (SSML), [http://www.w3.org/TR/speech-synthesis](http://www.w3.org/TR/speech-synthesis). A SALT browser may support any other speech synthesis formats. (Note: at the time of writing, the W3C SSML specification is currently a Working Draft and not yet a W3C Recommendation.)

The following example shows text with an instruction to emphasize certain key phrases:

```xml
<prompt id="giveBalance" xmlns:ssml="http://www.w3.org/2001/10/synthesis">
  You have <ssml:emphasis>five dollars</ssml:emphasis> left in your account.
</prompt>
```

2.1.1.2 value

The value element can be used to refer to text or markup held in elements of the document.

**value element**

**value:** Optional. Retrieves the values of an element in the document.

Attributes:

- **targetelement:** Required. The id of the element containing the value to be retrieved.
- **targetattribute:** Optional. The attribute of the element from which the value will be retrieved. If unspecified, no attribute is used, and the value defaults to the content (text or XML) of the element.

The targetelement attribute is used to reference an element within the containing document. The content of the element whose id is specified by targetelement is inserted into the text to be synthesized. If the desired content is held in an attribute of the element, the targetattribute attribute may be used to specify the necessary attribute on the targetelement. This is useful for referencing the values in HTML form controls, for example. In the following illustration, the value attributes of the txtBoxOrigin and txtBoxDest elements are inserted into the text before the prompt is output:

```xml
<prompt id="Confirm">
  Do you want to travel from
  <value targetelement="txtBoxOrigin" targetattribute="value" />
  to
  <value targetelement="txtBoxDest" targetattribute="value" />
  ?
</prompt>
```
2.1.3 content

The `content` element can be used to reference external content such as dynamically generated speech markup or remote audio files. It also holds optional inline content which will be rendered in the event of a problem with the externally referenced material. This can take the form of any of the inline content possible for `prompt`.

**content element**  
**content**: Optional. The `content` element specifies a link to an external output resource and identifies its type. SALT platforms should attempt to render if possible the content of the resource, but if this is impossible, any content specified inline will instead be output.

Attributes:
- **href**: Required. A URI referencing prompt output markup or audio.
- **type**: Optional. The media-type corresponding to the speech output format used. For XML content, typical types may be the W3C Speech Synthesis Markup Language format, specified as `application/ssml+xml`, or proprietary formats such as `application/x-sapitts+xml`. This attribute permits the SALT author to signal the format of a prompt resource and determine compatibility before a potentially lengthy download. Note, however, that it does not guarantee the format of the target (or inline resource), and platforms are free to treat the attribute (or its absence) in their own way. Formats required by SALT clients which support the basic media playback module (see section 2.8.1.5) are G.711 wav (audio/wav: 8kHz 8-bit mono [PCM] single channel) and headerless (audio/basic: 8kHz 8-bit mono [PCM] single channel). (Compression type is expected according to the telephony standard in the country of deployment, e.g. Mu-law in North American deployments, A-law in European deployments, etc.)

The following example holds one content element to reference XML content in SSML, and another to point to an audio file.

```xml
<prompt>
    <content href="/VoiceMailWelcome.ssml" type="application/ssml+xml" />
    After the beep, please record your message:
    <content href="/wav/beep.wav" />
</prompt>
```

2.1.4 Speech output configuration: `<param>`

Additional, non-standard configuration of the prompt engine is accomplished with the use of the `param` element, which passes parameters and their values to the platform. `param` is a child element of `prompt`.

The exact nature of the configurative parameters will differ according to the proprietary platform used. Values of parameters may be specified in an XML namespace, in order to allow complex or structured values.

**param element**  
**param**: Optional. Used to pass parameter settings to the speech platform.

param content

Attributes:
- **name**: required. The name of the parameter to be configured.
- **xmlns**: optional. Specifies a namespace and potentially a schema for XML content of the parameter.

So, for example, the following syntax:

```xml
<salt:param name="promptServer">//myplatform/promptServer</salt:param>
```

could be used to specify the location of a remote prompt engine for distributed architectures.

Note that page-level parameter settings in HTML profiles may also be defined using the `meta` element (see 2.8.2.2.1.5).
2.1.2 prompt attributes and properties

The prompt element holds the following attributes and properties. Attributes are supported by all browsers. Properties by uplevel browsers.

2.1.2.1 Attributes

- **id**: optional. The identifier of the prompt element. Must be a valid XML Name and unique within the document (i.e. of XML type ID).
- **bargein**: Optional. This Boolean flag indicates whether the platform is responsible for stopping prompt playback when speech or DTMF input is detected (this is sometimes also known as delegated bargein or cut-through). If true, the platform will stop prompt playback in response to input and flush the current subqueue. If false, the platform will take no default action in response to input. If unspecified, it defaults to true. In both cases the onbargein handler is called when input is detected (see section 2.1.4.2).
- **prefetch**: Optional. A Boolean flag which, if true, indicates to the platform that the external content of a prompt is likely to require a lengthy download, and may be prefetched sooner than playback time if possible. Defaults to false.
- **xmlns**: Optional. This is the standard XML namespacing mechanism and is used with inline XML prompts to declare a namespace and identify the schema of the format. See [http://www.w3.org/TR/REC-xml-names/](http://www.w3.org/TR/REC-xml-names/) for usage.
- **xml:lang**: Optional. String indicating the language of the prompt content. The value of this attribute follows the xml:lang definition in XML 1.0 ([http://www.w3.org/TR/REC-xml#sec-lang-tag](http://www.w3.org/TR/REC-xml#sec-lang-tag)). For example, xml:lang="en-US" denotes US English. The attribute is scoped, so if unspecified, a higher level element in the page may propagate the xml:lang value down to prompt (see equivalent in grammar element, section 2.2.1.1). If xml:lang is not specified at any level, the platform is free to determine the language of choice.

2.1.2.2 Properties

Uplevel browsers support the following properties in the prompt’s DOM object.

- **bookmark**: Read-only. A string object recording the text of the last synthesis bookmark encountered (see 2.1.4.1). For each playback, the property is set to null string until an onbookmark event is encountered.
- **status**: Read-only. Integer holding the status code returned by the speech platform on an event. The status property is only meaningful after status-setting events are thrown by the prompt object, and applications should examine it in the handler of the relevant event. A status code of zero is set by the oncomplete event (see 2.1.4.3). Other status values are set when the onerror event is thrown (see 2.1.4.4).

2.1.3 prompt methods

The queuing and playback of prompts may be controlled using the following methods on the prompt object. These methods will be supported by uplevel browsers. (Further manipulation of prompt subqueues is exposed through the PromptQueue object in uplevel browsers which support the PromptQueue module (see section 2.1.5)).

2.1.3.1 Queue

Queue the prompt onto the prompt subqueue. Takes an optional argument of type string (which may be markup, as for inline content). If no argument is provided, the method queues the inline content of the object. If an argument is provided, the value of the argument is treated as the string to be output instead of inline content, and is subject to any relevant features specified in the prompt’s attributes in section 2.1.2.1 (i.e. bargein, xmlns and xml:lang). After an individual prompt has finished normal playback, the oncomplete event is thrown, and its status code is set to zero.

**Syntax:**

This applies to whichever kind of input detection (or ‘bargein type’) is supported by platform. The type of detection could be set by using a platform-specific setting using the param element. It is not fired by keypress input on a visual display.

It is important to notice that even if bargein is false, the starting of a listen or dtmf object before the end of a prompt will still collect input immediately. If it is desired to begin collecting input only on completion of prompt playback, this sequence should be explicitly programmed (for example, in an uplevel HTML profile by wiring the prompt’s oncomplete event to the starting of listen/dtmf, or in a SMIL profile by declarative sequencing, and so on).
Object.Queue([strText]);

**Parameters:**
- **strText**: optional. String holding the text or markup to be sent to the speech output engine. If present, this value is used instead of the contents of the object. The content specified in the argument is treated exactly as if it were inline content in terms of resolving external references, etc.

**Return value:**
None.

**Exception:**
In the event of a problem with queuing the prompt, e.g. that external content cannot be retrieved and no alternate inline text is provided (see 2.1.1.3), the onerror event is thrown, and the prompt's status code is set to one of the relevant values described in 2.1.4.4.

Browsers should check prompts for freshness of content and validity of reference on the `Queue()` call. In general, error events should be raised as soon as possible after this call is made.

As noted in section 2.1.1.3, inline text can be specified as an alternative to external content, and this text is queued if external content is invalid or unretrievable. If no inline text is specified, and content is unable to be resolved, the onerror event is thrown as described above.

If `Queue()` is called in succession on multiple prompt objects, playbacks are queued in sequence onto a subqueue. Playback of the resulting subqueue does not begin until `Start()` is called (on a prompt or the PromptQueue object). If `Queue()` is called during playback (i.e. after PromptQueue.Start() but before the onempty event is thrown), the prompt is added to a new subqueue, which will only be played back after another explicit `Start()` call.

### 2.1.3.2 Start

Queue the prompt onto the prompt subqueue and schedule that subqueue for playback (i.e. begin playback immediately if no other prompts are currently in play, or begin playback of the subqueue directly after the last subqueue has ceased playback). Takes an optional argument of type string. If no argument is provided, the method queues the inline content of the object. If an argument is provided, the value of the argument is treated as the string to be output. This argument overrides any inline content, and is subject to any relevant features specified in the prompt's attributes in section 2.1.2.1 (i.e. bargein, xmlns and xml:lang). This method can be thought of a shorthand for `prompt.Queue([arg])` followed by a call to begin audio playback (PromptQueue.Start([arg]) in relevant profiles) and its content, arguments and the possible resulting events are just as if these two functions had been called sequentially.

**Syntax:**
Object.Start([strText]);

**Parameters:**
- **strText**: optional. String holding the text or markup to be sent to the speech output engine. If present, this value is used instead of the contents of the object.

**Return value:**
None.

**Exception:**
In the event of a problem with queuing or playing back the prompt, e.g. that external content cannot be retrieved and no alternate inline text is provided (see 2.1.1.3), the onerror event is thrown, and the prompt's status code is set to one of the relevant values described in 2.1.4.4.

### 2.1.4 prompt events

The prompt object supports the following events, whose handlers may be specified as attributes of the prompt element.

#### 2.1.4.1 onbookmark

Fires when a synthesis bookmark is encountered. Bookmarks are specified by application authors in the input to the speech output engine, and are used to notify an application that a particular point has been reached during playback. When the engine encounters a bookmark, the event is thrown to the platform. The example in section 2.6.2.2 shows how bookmarks in a prompt can be used to help determine the meaning of a user utterance.
On reception of this event, the **bookmark** property of the prompt object is set to the name of the bookmark thrown. The event does not pause or stop the playback.

### Event Object Information:

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>A bookmark in the rendered string is encountered</td>
</tr>
<tr>
<td>Default action</td>
<td>Returns the bookmark string</td>
</tr>
</tbody>
</table>

### Event Properties:

Although the event handler does not receive properties directly, the handler can query the event object for data.

#### 2.1.4.2 onbargein

Optional. Fires when an input event is detected from the user. This event corresponds to the detection of input from either speech or DTMF, and will be triggered by the onspeechdetected event on a started listen object (see section 2.2.4.3), or by the onkeypress event or onnoreco event on a started dtmf object for in-grammar and out-of-grammar keypresses respectively (sections 2.3.4.1, 2.3.4.3)

This handler is used to specify processing either (i) instead of, or (ii) in addition to the cessation of prompt playback on reception of an input event. (See section 3.2.1 for use of the **bargein** attribute to automatically stop prompt playback on detection of such an event.)

(i) If the **bargein** attribute is false and user input is detected, the prompt will keep playing when the onbargein event fires and while its associated processing is executed (unless of course it is explicitly stopped elsewhere in the application). This may be used in an email reader application, for example, where commands are enabled which do not require the prompt to stop (e.g. 'speak louder' or 'read faster') or for bookmarking (such as the example in 2.6.2.2).

(ii) If the **bargein** attribute is true, and user input is detected, the onbargein handler will fire after prompt playback has been halted by the platform and the prompt subqueue flushed. This may be used to specify any additional processing of a bargein event (e.g. to log the timing of the bargein).

It should not need restating that whether or not this event is specified, prompt playback is stopped and the subqueue flushed automatically by user input when the **bargein** attribute is set to true (section 3.2.1). (The automatic method generally results in less latency than using the onbargein handler to script an explicit PromptQueue.Stop())

### Event Object Information:

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>A speech/dtmf input event is encountered</td>
</tr>
<tr>
<td>Default action</td>
<td>None</td>
</tr>
</tbody>
</table>

### Event Properties:

Although the event handler does not receive properties directly, the handler can query the event object for data.

#### 2.1.4.3 oncomplete

This event fires when the prompt playback completes normally. It has no effect on the rest of the prompt subqueue. (In relevant profiles, after the oncomplete event of the last prompt in the subqueue, the onempty event is thrown by the PromptQueue object (section 2.1.5.3.1)).

### Event Object Information:

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>prompt playback completes</td>
</tr>
<tr>
<td>Default action</td>
<td>Set status = 0.</td>
</tr>
</tbody>
</table>

---

4 Note that in multimodal profiles where a visual display is used, the keypress event from a GUI element will not trigger the onbargein event.

5 As noted previously, it is important to remember that the starting of a listen or dtmf object before the end of a prompt will still collect input immediately. If it is desired to begin collecting input only on completion of prompt playback (i.e. that the onbargein event never fires on the prompt), this sequence should be explicitly programmed (for example, in an uplevel HTML profile by wiring the prompt's oncomplete event to the starting of listen/dtmf, or in a SMIL profile by declarative sequencing, and so on).
**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

2.1.4.4 *onerror*

The *onerror* event is fired if a serious or fatal error occurs with a prompt such that it is unable to be queued or played. The *onerror* event will typically be thrown after the `Queue()` command, and before playback of the current subqueue begins. Different types of errors are distinguished by status code and are shown in the event object information table below. The throwing of this event by a single prompt will flush the subqueue in which it is contained.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>To invoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>The synthesis process experiences a serious or fatal problem.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default action</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>On encountering an error, status codes are set as follows:</td>
<td></td>
</tr>
<tr>
<td><strong>status -1</strong>: Generic failure to queue the prompt onto the PromptQueue object.</td>
<td></td>
</tr>
<tr>
<td><strong>status -2</strong>: Failure to find a speech output resource (for distributed architectures)</td>
<td></td>
</tr>
<tr>
<td><strong>status -3</strong>: An illegal property/attribute setting that causes a problem with the synthesis request.</td>
<td></td>
</tr>
<tr>
<td><strong>status -4</strong>: Failure to resolve content – this is likely to be an unreachable URI, or malformed markup content.</td>
<td></td>
</tr>
</tbody>
</table>

**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

2.1.4.5 *Telephony hang-up*

In telephony profiles where the PromptQueue object is not supported, a disconnect event will have the effect of stopping playback of all prompts and flushing all subqueues.

2.1.5 *PromptQueue object*

The PromptQueue object is a browser object used to control prompt playback. It is accessible from script and has no markup element on the SALT page. Not all browsers need to support the PromptQueue module. In an HTML profile where the PromptQueue module is supported, the PromptQueue object will be a child of the `window` object.

The PromptQueue is maintained as a browser object rather than a markup element for two reasons:

1. It maintains a central object for playback control. The asynchronous nature of the call to `prompt.Queue()` means that with multiple queued prompts, the application cannot know (without explicit maintenance scripts) which prompt is currently being played back, and therefore which to pause/stop/resume, etc. when necessary. The PromptQueue object provides a single object for doing this.

2. This permits prompt playback to be uninterrupted across page transitions, since the PromptQueue object persists across the loading and unloading of individual pages. That is, unlike the markup elements in the DOM, it is not destroyed when its containing page is unloaded (although events which would otherwise be thrown to the prompt elements, are lost if the containing page has been unloaded). An example of this is shown in section 2.6.2.3.

The PromptQueue object operates in the following manner:

The PromptQueue is a singleton object which conceptually holds one or more prompt subqueues. As outlined in 2.1, these subqueues are delimited by `Start()` calls (either on individual prompts or on the PromptQueue itself). A new subqueue is initiated by calling `Queue()` on a prompt either when the PromptQueue is empty, or after a `Start()` call when it is playing back. That is, each set of prompts queued and followed by `Start()` is considered a single subqueue. A subqueue will not be played back until a `Start()` call is made.
It will be seen below that actions of the PromptQueue are local to a subqueue rather than the entire PromptQueue object, with the exception of the Flush() and Change() methods. The events supported by PromptQueue have scope at the level of subqueue. For example, the completion of playback of each subqueue is signaled by the onempty event (see 2.1.5.3.1). An onerror event on a single prompt will flush the subqueue in which that prompt was held, but will not affect any other subqueues in the PromptQueue (and therefore does not necessarily halt playback, see 2.1.5.3.2).

Bargein behavior is also local to a subqueue: recall from section 2.1.2.1 that where bargein is set true, an onbargein event on a single prompt will stop playback and flush the current subqueue. Similarly, all PromptQueue methods take effect in the scope of the subqueue rather than that of the PromptQueue object, except PromptQueue.Flush(), which is a global action to flush all the subqueues from the PromptQueue object, and PromptQueue.Change(), which applies adjustments of speed and volume to all subqueues on the PromptQueue.

For a detailed illustration of queuing and the PromptQueue, see 2.1.5.4.

The properties and methods of the PromptQueue object are described in detail below.

2.1.5.1 PromptQueue properties

Uplevel browsers support the following properties in the PromptQueue object.

- **status**: Read-only. Status code returned by the speech platform. The status code of zero indicates a successful completed subqueue operation by the speech platform, a negative status code indicates an error on the speech output platform.

2.1.5.2 PromptQueue methods

2.1.5.2.1 Start

Schedule an open subqueue of prompts for playback, that is, begin playback of the subqueue immediately if no other prompts are currently in play, or begin playback of the subqueue directly after the last subqueue on the PromptQueue has ceased playback. When the final prompt in a subqueue finishes playback (and after the throwing of that prompt's oncomplete event), an onempty event is thrown to the PromptQueue object (see 2.1.5.3.1) and its status property is set to zero. If no prompts are in the subqueue (including the case where all subqueues are already scheduled for playback), or a problem arises with the speech output resource, this call throws an onerror event with the error codes listed in 2.1.5.3.2.

**Syntax:**

```
PromptQueue.Start();
```

**Parameters:**

None.

**Return value:**

None.

**Exception:**

In the event of a problem the onerror event is fired, and the status code is set to a negative value, as listed in 2.1.5.3.2.

2.1.5.2.2 Pause

This method pauses playback of the current subqueue without flushing the audio buffer or otherwise affecting the subqueue. This method has no effect if playback is paused or stopped. Notice that Pause() is a synchronous method with a return value. While playback is paused, Start calls to playback (on a prompt or the PromptQueue) have the usual effects of subqueue delimitation and scheduling, but playback remains in a paused state (until Resume is called).

**Syntax:**

```
PromptQueue.Pause();
```

**Parameters:**

None.

**Return value:**

0 for successful pause, -1 for failure.

**Exception:**

None.
2.1.5.2.3 Resume
This method resumes playback after a pause. This method has no effect if playback has not been paused. Notice that Resume() is a synchronous method with a return value.

**Syntax:**
```
PromptQueue.Resume();
```

**Parameters:**
None.

**Return value:**
0 for successful resumption, -1 for failure.

**Exception:**
None.

2.1.5.2.4 Change
Change speed and/or volume of playback of prompts in all subqueues. Change() may be called before playback begins, or during playback. It takes effect beginning with the first or current prompt (as appropriate), and applies to prompts in all subqueues on the PromptQueue object. The adjustment factors to speed and volume are relative to current speed and volume, e.g. the consecutive commands:

```
PromptQueue.Change(2.0, 2.0);
PromptQueue.Change(2.0, 2.0);
```

will twice double the rates of speed and volume relative to those which were in effect before the first command was made.

**Syntax:**
```
PromptQueue.Change(speed, volume);
```

**Parameters:**
- **speed:** Required. The factor to change. speed=2.0 means double the current rate; speed=0.5 means halve the current rate; speed=1.0 means keep the current rate; speed=0 means to restore the default value.
- **volume:** Required. The factor to change. volume=2.0 means double the current volume, volume=0.5 means halve the current volume, volume=1.0 means keep the current volume; volume=0 means to restore the default value.

**Return value:**
None.

**Exception:**
If the Change() method is not supported, the onerror event is fired, and the status code is set to -3.

2.1.5.2.5 Stop
Stop playback and flush the subqueue. If playback has been paused, the method simply flushes the current subqueue. If playback is not underway at all, the method has no effect.

**Syntax:**
```
PromptQueue.Stop();
```

**Parameters:**
None.

**Return value:**
None.

**Exception:**
There are no explicit exceptions associated with this method.

2.1.5.2.6 Flush
Stop playback and flush the audio buffer. All subqueues are removed from the PromptQueue. If playback is not underway (i.e. it has not been started or has already been stopped or paused) the method flushes all prompt subqueues. In telephony profiles, the Flush() method is executed automatically by the detection of a disconnect (the order of firing is as follows: listen, dtmf, PromptQueue), and the onerror event is thrown, with status code -30.

**Syntax:**
PromptQueue.Flush();

**Parameters:**
None.

**Return value:**
None.

**Exception:**
There are no explicit exceptions associated with this method. However, onerror is thrown and status code -30 set in telephony profiles when a disconnect event automatically invokes Flush().

### 2.1.5.3 PromptQueue event handlers

#### 2.1.5.3.1 onempty
This event fires when the last of the prompts in a subqueue have finished playback. It fires after the oncomplete is fired on the last prompt of the subqueue (and therefore only fires when the playback of the prompt subqueue completes naturally without explicit stop calls). For prompt queues which are not stopped by other means, there will be one onempty event for every subqueue, i.e. for every Start() call made.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Final prompt in prompt subqueue has completed playback.</td>
</tr>
<tr>
<td>Default action</td>
<td>Set status = 0 if playback completes normally.</td>
</tr>
</tbody>
</table>

**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

#### 2.1.5.3.2 onerror
The onerror event is fired if a serious or fatal error occurs with the synthesis (voice output) process. Since onerror on the PromptQueue object will fire on generic platform errors, playback is stopped on reception of this event and the subqueue is flushed. (Other queues held in the PromptQueue object are not affected.) For platform errors, a status code of -1 is set; for errors fired due to Start() being called on an empty subqueue, a status code of -2 is set, and for Change() when unsupported, a status code of -3 is set. onerror is also fired in telephony profiles as a result of the automatic call to Flush() on detection of a disconnect (status -30), or if playback is attempted after a disconnect event.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>The synthesis process experiences a serious or fatal problem.</td>
</tr>
<tr>
<td>Default action</td>
<td>On encountering an error, status codes are set as follows: status -1: A generic speech output resource error occurred during playback. status -2: the Start() call was made on an empty prompt subqueue. status -3: the Change() call was made, but the platform does not support it. status -30: (telephony profiles only) a disconnect invoked the Flush() method, or prompt playback was attempted after disconnect.</td>
</tr>
</tbody>
</table>

**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

### 2.1.5.4 PromptQueue illustrations

This section contains diagrams illustrating the mechanisms of the PromptQueue in action.

#### 2.1.5.4.1 Initializing a subqueue
Figure 1: single prompt queued

Figure 1 shows the results of queuing a single prompt, `prompt1`, using its `Queue` method. This initializes a new subqueue, and places `prompt1` at its head.

2.1.5.4.2 Adding a second prompt to the subqueue

Figure 2: two prompts queued

Figure 2 shows the queuing of a second prompt, `prompt2`, again with the `Queue` method. It is added to the tail of the same subqueue, after `prompt1`.

2.1.5.4.3 Adding a third prompt to the subqueue

Figure 3: three prompts queued

Figure 3 shows the queuing of a third prompt, `prompt3`, using its `Queue` method. As with `prompt2`, it is added to the tail of the same subqueue.

2.1.5.4.4 Call to begin playback
Figure 4: Starting playback

Figure 4 shows the call to start playback, `PromptQueue.Start()`, on the prompts currently queued. The call has the effect of (i) closing the subqueue after `prompt3`, and (ii) scheduling the subqueue for playback, beginning with the first prompt, `prompt1`. Since no other subqueues are being output, playback begins immediately. When `prompt1` has finished, its `oncomplete` event will be fired, and the next prompt in the subqueue will begin (`prompt2`).

2.1.5.4.5 Prompt queuing during playback

Figure 5 shows a prompt being queued while playback is underway. Since the last subqueue was closed, the new prompt, `prompt4` initializes a new subqueue and places itself at the head. This happens while the current subqueue is being played back. (As above, when `prompt2` has finished, its `oncomplete` event will be fired, and the next prompt in the subqueue will begin (`prompt3`).

2.1.5.4.6 Playback using Start on the prompt

Figure 6 shows the queuing of another prompt and a call to playback on the new subqueue. Two things are important to note here.

Firstly, the `Start` method on the `prompt` is used, rather than the `Start` method on the `PromptQueue` object. The calling of `Start` on the new prompt, `prompt5`, is equivalent in this profile to the two consecutive commands `prompt5.Queue` followed by `PromptQueue.Start` (as described in section 2.1.3.2). The effect of the call is as follows:
(i) prompt5 is queued onto the tail of the new subqueue; (ii) the new subqueue is closed after prompt5; and (iii) the new subqueue is scheduled for playback on completion of the current subqueue.

Secondly, the call to playback happens before the first subqueue has finished playback. So the second subqueue is scheduled for output as soon as the first is finished, that is, prompt4 will begin playback immediately after prompt3 has completed.

2.1.5.4.7 A third subqueue

Figure 7: another new subqueue

Figure 7 shows the creation of another new subqueue. The very first subqueue (consisting of prompt1, prompt2 and prompt3) has now been played out and has disappeared from the PromptQueue. The current subqueue is now the second subqueue, of which prompt4 is currently in playback. The queuing of another prompt, prompt6, initializes the third subqueue and places prompt6 at its head.

2.1.5.4.8 End of the second subqueue

Figure 8: completion of current subqueue

Figure 8 shows the final prompt of the current subqueue, prompt5, being played out. When it has finished, its oncomplete event will be fired, and since it ends a subqueue, the onempty event will then be fired on the PromptQueue object. prompt6 awaits in the next subqueue.

2.1.5.4.9 Subqueue awaiting call to playback

Figure 9: subqueue awaiting Start call
Figure 9 shows the **PromptQueue** after the completion of `prompt5`. The second subqueue, made up of `prompt4` and `prompt5`, has played out and is no longer on the **PromptQueue**. The current subqueue is now made up of `prompt6`, which is still open and has not begun playback (since no `Start` call, either on a `prompt` or on the **PromptQueue**, has yet been made).

### 2.1.5.4.10 Two subqueues before a Stop call

![Figure 10: two subqueues before Stop()](Image)

Figure 10 shows a different scenario: two subqueues on the **PromptQueue**, right before a `Stop()` call. `promptA` and `promptB` are in the current subqueue, and `promptA` is in playback. `promptY` and `promptZ` are in the second subqueue, which has not yet been scheduled for playback.

### 2.1.5.4.11 Result of Stop, next subqueue not yet scheduled

![Figure 11: result of Stop, next subqueue not yet scheduled](Image)

Figure 11 shows the results of a `Stop()` call on the situation in Figure 10. The prompt that was in playback, `promptA`, was stopped and its subqueue flushed (including `promptB`). (Recall that a user input event during a prompt whose `bargein` attribute is true will trigger the same behavior.) The next subqueue, consisting of `promptY` and `promptZ`, remains on the **PromptQueue**, awaiting a call to begin playback.

This situation is effectively a return to the situation in Figure 2, where two prompts are on a subqueue that has not yet been scheduled for playback. Should **PromptQueue.Start()** be called, playback will begin on the existing `promptY` and `promptZ` subqueue. Should **PromptQueue.Stop()** be called again, it will have no effect, since the subqueue is not in playback. Should another prompt be queued, it will be added to the tail of the existing `promptY` and `promptZ` subqueue.

### 2.1.5.4.12 Two subqueues before a Stop call, both scheduled
Figure 12: Two subqueues before Stop, both scheduled

Figure 12 shows a different scenario: two subqueues on the PromptQueue, right before a Stop() call. promptA and promptB are in the current subqueue, and promptA is in playback. promptY and promptZ are in the second subqueue, and have been scheduled for playback (e.g. PromptQueue.Start was called after they were queued).

2.1.5.13 Result of Stop, next subqueue already scheduled

Figure 13: Result of Stop, next subqueue already scheduled

Figure 13 shows the results of a Stop() call on the situation in Figure 12. Again, the prompt that was in playback, promptA, was stopped and its subqueue flushed (including promptB). The next subqueue, consisting of promptY and promptZ, this time begins playback immediately, since it was already scheduled for playback.

2.1.5.14 Result of Flush

Figure 14: result of Flush()

Figure 14 shows the results of the Flush() method on the situation in Figure 10, and, in fact, on any situation. The Flush() method stops playback and flushes all prompts in all subqueues out of the PromptQueue, leaving it empty. (Recall also that in telephony profiles this method is called automatically when a disconnect event occurs.)

2.1.5.15 Pause() and the queuing model

As described in 2.1.5.2.2, the PromptQueue.Pause() command can be used to stop playback without flushing the audio buffer, ready to resume playback at the point at which it was halted. Since the semantics of this command is to temporarily cease playback without otherwise affecting the subqueues on the PromptQueue, this allows multiple queuing operations to be performed while playback is paused. So a Start call, either on a prompt or the PromptQueue, made while playback is paused will still have the usual semantics of delimiting a subqueue, and scheduling a subqueue for
playback, even though playback is not underway. (This permits the scheduling of multiple subqueues 'under the covers', i.e. without corresponding immediate execution of playback, if this is desired.)

### 2.2 Speech input: <listen>

The listen element is used for speech recognition, for audio recording, or for both.

A listen element which is used for speech recognition contains one or more grammar elements, which are used to specify possible user inputs. A listen element which is used for audio recording contains a record element which is used to configure the recording process. A listen element used for simultaneous recognition and recording holds one or more grammar elements and a record element. In all cases, bind can be used to process the results obtained from recognition and/or recording.

Many of the features of the listen object are used in both the recognition and the recording scenarios, and the attributes, properties, methods and event handlers of listen have similar behavior whether it is used for speech recognition or for recording. In those browsers which support the Concurrent Recognition and Recording module (see 2.8.1.4), simultaneous speech recognition and audio recording is enabled with the specification of both grammar and record in a single listen object, and the behavior of the object is driven by speech recognition events.

listen elements used for speech recognition may also take a particular mode - ‘automatic’, ‘single’ or ‘multiple’ – to distinguish the kind of recognition scenarios which they enable and the behavior of the recognition platform, as described in 2.2.6.

The use of the listen element for speech recognition is defined in sections 2.2.1 to 2.2.7. The use of the listen element for recording is described in detail in section 2.2.8.

#### 2.2.1 listen content

As noted above, the listen element contains one or more grammars (and/or a record element), and (optionally) a set of bind elements which inspect the results of the speech input and copy the relevant portions to values in the containing page. It also permits further configuration using the param mechanism.

In uplevel browsers, listen also supports the programmatic activation and deactivation of individual grammar rules.

#### 2.2.1.1 <grammar> element

The grammar element is used to specify grammars, either inline or referenced using the src attribute. grammar is a child element of listen. At least one grammar (either inline or referenced) must be specified for speech recognition. Inline grammars must be text-based grammar formats, while referenced grammars can be text-based or binary type. Multiple grammar elements may be specified, in which case each grammar element is considered in a separate namespace for the purpose of grammar compilation. All the grammars of a listen element are treated as active unless (i) explicitly deactivated, or (ii) inactive by virtue of internal content.

To enable interoperability of SALT applications, SALT browsers must support the XML form of the W3C Recommendation for Speech Recognition Grammar Specification (SRGS), [http://www.w3.org/TR/speech-grammar/](http://www.w3.org/TR/speech-grammar/). A SALT browser may support any other grammar formats. (Note: at the time of writing, the W3C SRGS specification is not yet a W3C Recommendation.) In order to guarantee complete interoperability of grammars, it is expected that W3C will eventually require the use of the W3C Semantic Interpretation (SI) specification with W3C grammars ([http://www.w3.org/TR/semantic-interpretation/](http://www.w3.org/TR/semantic-interpretation/)). Until this is the case, SALT platforms which implement semantic interpretation using W3C grammar formats should also support W3C SI.

**Attributes:**

- **name** Optional. This value identifies the grammar for the purposes of activation and deactivation (see 2.2.3.4 and 2.2.3.5). Grammars within the same listen element must not be identically named. Note that the use of name does not enable the referencing of the rules of one inline grammar from another.
- **src** Optional. URI of the grammar to be included. The reference of the URI must be a valid grammar reference or grammar rule reference according to the semantics of the grammar format used. Specification of the src attribute in addition to an inline grammar is illegal and will result in an invalid document.
- **type** Optional. For externally referenced grammars, the media-type corresponding to the grammar format used. This may refer to text or binary formats. Typical types may be the W3C XML grammar format, specified as
application/srgs+xml, or proprietary formats such as application/x-sapibinary. The type attribute permits the SALT author to signal the format of a grammar resource and determine compatibility before a potentially lengthy download. However, note that it does not guarantee the format of the target (or inline resource), and platforms are free to treat the attribute (or its absence) in their own way. If unspecified, the type will default to the common format required for interoperability.

- **xmlns** Optional. This is the standard XML namespacing mechanism and is used with inline XML grammars to declare a namespace and identify the schema of the format. See [http://www.w3.org/TR/REC-xml-names/](http://www.w3.org/TR/REC-xml-names/) for usage.

- **xml:lang** Optional. String indicating which language the grammar refers to. The value of this attribute follows the xml:lang definition in XML 1.0 ([http://www.w3.org/TR/REC-xml#sec-lang-tag](http://www.w3.org/TR/REC-xml#sec-lang-tag)). For example, xml:lang="en-US" denotes US English. The attribute is scoped, so if unspecified, a higher level element in the page may propagate the xml:lang value down to grammar (e.g. listen). If xml:lang is specified in multiple places then xml:lang follows a precedence order from the lowest scope – remote grammar file (i.e xml:lang may be specified within the grammar file) followed by grammar element followed by listen element, so for external grammars, it may even be overridden by xml:lang specified within the target grammar. If xml:lang is completely unspecified, the platform is free to determine the language of choice.

Whether inline or referenced, SALT grammars are expected to respect the declaration and referencing semantics of the format used.

### Notes for use of W3C SRGS grammars
For applications using W3C SRGS grammars the following should be noted.

A W3C grammar used inline should declare a root. For a W3C grammar used by reference, the src attribute may include the rulename fragment. If the src does not reference a public rule of the grammar then the reference is in error. If the rulename fragment is omitted then the reference is an implied reference to the root rule of the referenced grammar. If the referenced grammar has no root then the src reference is in error. It is legal to have more than one reference to the same external grammar where each grammar element references a different public rulename of that grammar (by the rulename fragment).

### Example referenced and inline grammars

```xml
<salt:grammar src="cities.grxml" type="application/srgs+xml" />
```

or

```xml
<salt:grammar xmlns="http://www.w3.org/2001/06/grammar">
  <grammar root="root">
    <rule id="root">
      <item repeat="0-1">from </item>
      <ruleref name="#cities" />
    </rule>
    <rule id="cities">
      <one-of>
        <item> Cambridge </item>
        <item> Seattle </item>
        <item> London </item>
      </one-of>
    </rule>
  </grammar>
</salt:grammar>
```

The specification of both the src attribute and inline content in the same grammar element will result in an invalid document.

### Grammar types

---

6 xml:lang is a 'global' XML attribute which when placed on an element, says that any human language used in that element and all elements beneath it, is in the language referred to by xml:lang.
SALT grammars are expected to be either context-free grammars (CFGs), as illustrated above and commonly used today in command driven telephony voice applications, or N-Gram grammars, as used in larger vocabulary dictation and "How can I help you?"-style applications. Whereas listens of automatic and single mode can be used with CFG or N-Gram grammars (or both), listens of 'multiple' mode will typically use N-Grams to accomplish dictation. (For the mode attribute on listen, see section 2.2.6).

To enable interoperability of SALT applications, SALT browsers which support N-Gram recognition must support the W3C Recommendation for Stochastic Language Models (N-Gram) (http://www.w3.org/TR/ngram-spec). A SALT browser may support any other stochastic grammar formats. (Note: at the time of writing, the W3C N-Gram specification is currently a Working Draft and not yet a W3C Recommendation.)

In terms of the recognition result, a listen using N-Grams will hold the recognized text or the N-Best variants in its XML result structure, which may take the form of a word graph.

2.2.1.2 <bind> element

The bind element is used to bind values from spoken input into the page, and/or to call methods on page elements. bind is a child element of listen.

The input result processed by the bind element is an XML document containing a semantic markup language (e.g. W3C Natural Language Semantic Markup Language) for specifying recognition results. Its contents typically include semantic values, actual words spoken, and confidence scores. The return format could also include alternate recognition choices (as in an N-best recognition result).

To enable interoperability of SALT applications, SALT browsers must support the W3C Recommendation for Natural Language Semantic Markup Language (NLSML) format (http://www.w3.org/TR/nl-spec/). A SALT browser may support any other semantic markup language. (Note: at the time of writing, the W3C NLSML specification is currently a Working Draft and not yet a W3C Recommendation.)

A sample W3C NLSML return for the utterance "I'd like to travel from Seattle to Boston" is illustrated below:

```xml
<result grammar="http://flight" xmlns:xf="http://www.w3.org/2000/xforms">
  <interpretation confidence="0.4">
    <input mode="speech">
      I'd like to travel from Seattle to Boston
    </input>
    <xf:instance>
      <airline>
        <origin_city confidence="0.45">Seattle</origin_city>
        <dest_city confidence="0.35">Boston</dest_city>
      </airline>
    </xf:instance>
  </interpretation>
</result>
```

Since a recognition result produces an XML document, the values to be bound from that document are referenced using an XPath query. And since the elements in the page into which the values will be bound should be uniquely identified (they are likely to be form controls), these target elements are referenced directly with the targetelement attribute.

The binding operation is executed whenever a recognition result is returned and before the relevant recognition event is thrown. When bind is used for assignment, the result of the XPATH query is copied from the result DOM into the page DOM. If the target of assignment targetattribute is of type string, the result will be converted into a well-formed XML string without loss of information. This feature can be used with a complete recognition result, for instance, to submit the entire result to a web server. Otherwise, if the targetattribute is of type XML DOM Node, the assignment follows the copy-of semantics of XSLT 1.0 (defined at http://www.w3.org/TR/xslt#copy-of), namely, the DOM node tree returned by the XPATH will be copied to the targetattribute as a DOM node tree. It raises no events itself. If it fails to execute or contains errors in content, no operation is performed.

Attributes:
- targetelement: Required. The name of the element to which the value content from the recognition XML will be assigned (as in W3C SMIL 2.0).
• **targetattribute**: Optional. The attribute of the target element to which the value content from the recognition XML will be assigned (as with the attributeName attribute in SMIL 2.0). If unspecified, defaults to "value".

• **targetmethod**: Optional. The method of the target element which will be called if the bind is executed. Such methods are presently limited to functions that assume "void" for both the argument list and the return type. Examples include the submit method of the HTML form object, the click method of the button and the hyperlink objects, and the Start and Stop methods of a listen object.

• **test**: Optional. String holding an XML pattern (as for the test attribute of conditional expressions in XSLT, http://www.w3.org/TR/xslt#section-Conditional-Processing), indicating the condition under which the bind will be executed. If unspecified, no condition is applied and the bind element will always be executed on the return of recognition results.

• **value**: Optional. An XPath (as in http://www.w3.org/TR/xpath) string that specifies the value from the recognition result document to be assigned to the target element. Ignored when used with method execution (targetmethod). If unspecified and used with assignment, defaults to the entire recognition result document.

Each bind directive can have at most one targetmethod or targetattribute attribute. Specification of more than one, or of both targetattribute and targetmethod will result in an invalid document.

When multiple bind directives return a Boolean value "true" on their respective test conditions, they are executed in document order.

**Example:**
So given the recognition result of the examples above, the following listen element uses bind to transfer the values in origin_city and dest_city into the target page elements txtBoxOrigin and txtBoxDest:

```html
  ...
  <form id="formTravel">
    <input name="txtBoxOrigin" type="text"/>
    <input name="txtBoxDest" type="text"/>
  </form>
  ...
  <salt:listen id="listenTravel">
    <salt:grammar src="./city.grxml"/>
    <salt:bind targetelement="txtBoxOrigin" value="//origin_city"/>
    <salt:bind targetelement="txtBoxDest" value="//dest_city"/>
  </salt:listen>
  ...
</html>
```

This binding may be conditional, as in the following example, where a test is made on the confidence attribute of the dest_city result as a pre-condition to the bind operation:

```html
<salt:bind targetelement="txtBoxDest" value="//dest_city" test="//dest_city[@confidence > 0.4]"/>
```

The bind element is also able to call methods on the specified element, so the following example would submit the HTML travel form without needing any script code:

```html
<salt:bind test="//dest_city[@confidence > 0.4]" targetelement="formTravel" targetmethod="submit"/>
```

It is important to remember that many speech recognizers return results as a set of N-Best alternatives within the recognition XML. In these cases, since a typical XPath query may return more than one node, the extraction of a single result may require an array index to identify the most likely node (e.g. "(//dest_city)[1]" for the query in the example on this page). For the purpose of illustrative simplicity, the examples in the rest of this document assume a single relevant node in the recognition result.

---

7 It is important to remember that many speech recognizers return results as a set of N-Best alternatives within the recognition XML. In these cases, since a typical XPath query may return more than one node, the extraction of a single result may require an array index to identify the most likely node (e.g. "(//dest_city)[1]" for the query in the example on this page). For the purpose of illustrative simplicity, the examples in the rest of this document assume a single relevant node in the recognition result.
The `bind` element is a simple declarative means of processing recognition results on downlevel or uplevel browsers. For more complex processing, the `listen` DOM object supported by uplevel browsers implements the `onreco` (or `onnoreco`) event handler to permit programmatic script analysis and post-processing of the recognition return (see 2.2.4.1) or recording results (see 2.2.8.4.2).

Further illustrations of the use of `bind` may be found in the sample markup examples in 2.6.

### 2.2.1.3 Recording: `<record>`

The recording of audio is described in section 2.2.8.

### 2.2.1.4 Speech recognition configuration: `<param>`

Additional, non-standard configuration of the speech recognition engine is accomplished with the use of the `param` element which passes parameters and their values to the platform. `param` is a child element of `listen`.

The exact nature of the configurative parameters will differ according to the proprietary platform used. Values of parameters may be specified in an XML namespace, in order to allow complex or structured values.

**param element**

- **param**: Optional. Used to pass parameter settings to the speech platform.

**param content**

Attributes:

- **name**: required. The name of the parameter to be configured.
- **xmlns**: optional. Specifies a namespace and potentially a schema for XML content of the parameter.

So, for example, the following syntax:

```
<salt:param name="recoServer">//myplatform/recoServer</salt:param>
```

could be used to specify the location of a remote speech recognition server for distributed architectures.

Note that in HTML profiles, page-level parameter settings may also be defined using the `meta` element (see 2.8.2.2.1.5).

### 2.2.2 `listen` attributes and properties

The following attributes are supported by all browsers and the following properties are supported by uplevel browsers.

#### 2.2.2.1 Attributes

The following attributes of `listen` are used to configure the speech recognizer for a dialog turn.

- **id**: optional. The identifier of the `listen` element. Must be a valid XML Name and unique within the document (i.e. of XML type ID).
- **initialtimeout**: Optional. The time in milliseconds between the start of recognition (if no prompt is in playback) or the end of prompt (if a prompt is in playback) and the detection of speech. This value is passed to the recognition platform, and if exceeded, an `onsilence` event will be thrown from the recognition platform (see 2.2.4.2). A value of 0 effectively disables the timeout. If the attribute is not specified, the speech platform will use a default value.
- **babbletimeout**: Optional. The maximum period of time in milliseconds for an utterance. For `listens` in automatic and single mode (see 2.2.6), this applies to the period between speech detection and the speech endpoint or `Stop()` call. For `listens` in ‘multiple’ mode, this timeout applies to the period between each speech detection and subsequent phrase recognition—i.e. the period is restarted after each return of results or other event. If exceeded, the `onnoreco` event is thrown with status code -15. This can be used to control when the recognizer should stop processing excessive audio. For automatic mode `listens`, this will happen for exceptionally long utterances, for example, or when background noise is mistakenly interpreted as continuous speech. For single mode `listens`, this may happen if the user keeps the audio stream open for an excessive amount of time (e.g. by holding down the stylus in tap-and-talk). For a summary of `onnoreco`
Speech Application Language Tags (SALT) 1.0

status codes, see section 2.2.4.4. A value of 0 effectively disables the timeout. If the attribute is not specified, the speech platform will use a default value.

- **maxtimeout**: Optional. The period of time in milliseconds between the call to start recognition if no prompt is in playback) or the end of prompt (if a prompt is in playback) and the returning of results to the browser. If exceeded, an onerror event is thrown – this caters for network or recognizer failure in distributed environments. For `listens` in 'multiple' mode, as with babbletimeout, the period is restarted after the return of each recognition or other event. Note that the maxtimeout attribute should be greater than or equal to the sum of initialtimeout and babbletimeout (and endsilence for automatic mode). A value of 0 effectively disables the timeout. If the attribute is not specified, the speech platform will use a default value.

- **endsilence**: Optional. For `listens` in automatic mode (see 2.2.6), the period of silence in milliseconds after the end of an utterance which must be free of speech after which the recognition results are returned. The speech recognizer may ignore this attribute for `listens` of modes other than automatic. If unspecified, defaults to platform internal value.

- **reject**: Optional. The confidence threshold for recognition rejection, below which the platform will throw the onnoreco event. If not specified, the speech platform will use a default value. Confidence scores are floating point values between 0 and 1. reject values lie in between.

- **xml:lang**: Optional. String indicating which language the speech recognizer should attempt to recognize. The string format follows the xml:lang definition in XML 1.0 (http://www.w3.org/TR/REC-xml#sec-lang-tag). For example, xml:lang="en-US" denotes US English. This attribute is only meaningful when xml:lang is not specified in the grammar element (see 2.2.1.1), or in its content.

- **mode**: Optional. String specifying the recognition mode to be followed (see 2.2.6 below). If unspecified, defaults to "automatic" mode.

Many of these attributes are used in the same way to configure the audio recording process, as detailed in section 2.2.8.2.1.

In certain HTML profiles, the HTML attributes accesskey and style may also be used as attributes of `listen`, as described in 2.8.2.1.1.

### 2.2.2.2 Properties

The following properties contain the results returned by the recognition process (these are supported by uplevel browsers).

- **recoresult** Read-only. The results of recognition, held in an XML DOM node object containing the recognition return, as described in 2.2.1.2. In case of no recognition, the return may be empty.

- **text** Read-only. A string holding the text of the words recognized. A SALT browser must attempt to extract such a string from the return result. Where the result format is known to the browser, this is found in a standard query related to that format, e.g. for NLSML, /result/interpretation[1]/input. For N-Best results, the string holds the text of the first (typically the most likely) utterance in the N-Best list. If the browser is unable to determine the text of the utterance (even after applying the standard query for the default format) the value will be null string.

- **status**: Read-only. Integer holding a status code returned by the recognition platform. The status property is only meaningful after status-setting events are thrown by the listen object, and applications should examine it in the handler of the relevant event. Possible values are 0 for successful recognition, or the failure values -1 to -9 (as defined in the exceptions possible on the Start method (section 2.2.3.1) and Activate method (section 2.2.3.4)) and statuses -11 to -15 set on the reception of recognizer error events (see 2.2.4.5), statuses -20 to -24 in the case of recording (see 2.2.8.4.5), and status -30 for telephony hang-ups (see 2.2.4.5).

### 2.2.3 listen methods

The execution of listen elements may be controlled using the following methods in the listen's DOM object. With these methods, browsers can start and stop listen objects, cancel recognitions in progress, and uplevel browsers can also activate and deactivate individual grammar top-level rules.

---

8 This may be interpreted by the platform as an instruction to allow maximum rather than infinite length input.
2.2.3.1 Start

The **Start** method starts the recognition process, using as active grammars all those which have not been explicitly deactivated (or are inactive by declaration). As a result of the **Start** method, a speech recognition event such as onreco, onnoreco, or onsilence will typically be fired, or an onerror event will be thrown in the case of an application or platform error. See section 2.2.4 for a description of these events. (Note that for telephony profiles, associated dtmf recognition events can also end the execution of listen, as described in 2.3.6.)

**Syntax:**
```
Object.Start()
```

**Parameters:**
None.

**Return value:**
None.

**Exception:**
The method sets a non-zero status code and fires an onerror event if it fails. The onerror event description in section 2.2.4.5 lists possible non-zero status codes.

On the calling of the **Start()** method the speech recognition platform must ensure that the active grammars of a listen are complete and up-to-date. Only one listen object may be started at a given time. If **Start()** is called on a listen object which is already in execution, the call has no effect. If **Start()** is called on a listen object while another is in execution, the onerror event is thrown on the object on which the second **Start()** was attempted. onerror events resulting from the **Start()** method are thrown according to the status codes in section 2.2.4.5.

2.2.3.2 Stop

The **Stop** method is a call to end the recognition process. The listen object stops processing audio, and the recognizer returns recognition results on the audio received up to the point where recording was stopped. Once the recognition process completes, all the recognition resources used by listen are released. The result of calling **Stop()** will be an onreco or onnoreco event and the return of a recognition result, or an onerror event. (Note that this method need not be used explicitly for typical recognitions in automatic mode (see 2.2.6), since the recognizer itself will stop the listen object on endpoint detection after recognizing a complete grammar match.) If the listen has not been started, the call has no effect. In telephony profiles, the **Stop()** method is executed automatically by the detection of a disconnect (the order of firing is as follows: listen, dtmf, PromptQueue), and as a result onreco or onnoreco is thrown.

**Syntax:**
```
Object.Stop()
```

**Parameters:**
None.

**Return value:**
None.

**Exception:**
There are no explicit exceptions associated with this method. However, if **Stop()** is called before speech is detected, the onnoreco event is fired and status code is set to -11 (as in 2.2.4.4), and if there is any problem an onerror event is fired with and the status codes as outlined in section 2.2.3.1 are set.

2.2.3.3 Cancel

The **Cancel** method stops the audio feed to the recognizer and releases recognizer resources. The platform may return a recognition result for a cancelled recognition (although this may be empty). If the recognizer has not been started, the call has no effect. No event is thrown when the **Cancel** method is called.

**Syntax:**
```
Object.Cancel()
```

**Parameters:**
None.

**Return value:**
None.

**Exception:**
None.
2.2.3.4 Activate

The Activate method activates the grammars of a listen. The first argument identifies the grammar for activation, the optional second argument identifies a top-level rulename within that grammar. If called during a 'started' listen, the change will not take effect until the listen is restarted. (Recall also that Activate() is not necessary in the default case: the grammars of a listen object are treated as active unless explicitly deactivated.)

Syntax:
Object.Activate(grammarName, [ruleName]);

Parameters:
- grammarName: Required. Name of the grammar (i.e. the name attribute of the relevant grammar).
- ruleName: Optional. Rule name within the grammar.

Return value:
None.

Exception:
There are no explicit exceptions associated with this method. However, if the grammar identified with the grammarName argument does not exist, an onerror event is fired and a value of -6 set in the status property of the listen object. (Note also that onerror would be fired as a result of the listen.Start() method if the rule identified by the ruleName argument does not exist.)

Note that for W3C SRGS grammars the rule name is not necessary (since only a single rule can be root).

2.2.3.5 Deactivate

The Deactivate method deactivates the grammars of a listen. The first argument identifies the grammar for deactivation, the optional second argument identifies a top-level rulename within that grammar. If called during a 'started' listen, the change will not take effect until the listen is restarted. If the grammar or rule is already deactivated, the call has no effect.

Syntax:
Object.Deactivate(grammarName, [ruleName]);

Parameters:
- grammarName: Name of the grammar (i.e. the name attribute of the relevant grammar).
- ruleName: Optional. Rule name within the grammar.

Return value:
None.

Exception:
There are no explicit exceptions associated with this method. However, if the grammar identified with the grammarName argument does not exist, an onerror event is fired and a value of -6 is set in the status property of the listen object. (Note also that onerror would be fired as a result of the listen.Start() method if the rule identified by the ruleName argument does not exist.)

Note that for W3C SRGS grammars the rule name is not necessary (since only a single rule can be root).

2.2.4 listen events

The listen object supports the following events, whose handlers may be specified as attributes of the listen element. For a graphical summary of events along the timeline in different modes of recognition see section 2.2.6.

It is important to notice that the recoresult property is updated for both successful and unsuccessful events from the speech recognizer. So applications should assume that the property holds a valid result from the user only in the case of successful recognitions. In the case of unsuccessful or aborted recognitions, the result may be an empty document, or it may hold extra information which applications are free to use or ignore. In either case, applications examining the recoresult property should do so in the relevant speech event handler, i.e. onreco or onnoreco, since the property may not be valid for examination at other times.

2.2.4.1 onreco

This event is fired when the recognizer has a successful recognition result available for the browser. This corresponds to a valid match in the grammar and a confidence value above the reject threshold. For listens in automatic mode, this event stops the recognition process automatically and clears resources. The onreco
Speech Application Language Tags (SALT) 1.0

handler is typically used for programmatic analysis of the recognition result and processing of the result into the page.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>User says something</td>
</tr>
<tr>
<td>Default action</td>
<td>Return recognition result object. In telephony profiles, status codes are set as follows:</td>
</tr>
<tr>
<td></td>
<td>status -30: (telephony profiles only): Stop() invoked by disconnect.</td>
</tr>
</tbody>
</table>

**Event Properties:**

Although the event handler does not receive properties directly, the handler can query the event object for data (see the use of the event object in the example below).

**Example**

The following XHTML fragment uses onreco to call a script to parse the recognition outcome and assign the values to the proper fields.

```html
...
<input type="button" value="Talk to me" onClick="listenCity.Start()" />
<input name="txtBoxOrigin" type="text" />
<input name="txtBoxDest" type="text" />
...
<salt:listen id="listenCity" onreco="processCityRecognition()">
<salt:grammar src="/grammars/cities.grxml" />
</salt:listen>
<script><![CDATA[
function processCityRecognition () {
    smlResult = event.srcElement.recoresult;
    origNode = smlResult.selectSingleNode("//origin_city/text()")
    if (origNode != null) txtBoxOrigin.value = origNode.value;
    destNode = smlResult.selectSingleNode("//dest_city/text()")
    if (destNode != null) txtBoxDest.value = destNode.value;
}
]]></script>
</html>
```

2.2.4.2 onsilence

**onsilence** handles the event of no speech detected by the recognition platform before the duration of time specified in the initialtimeout attribute on the listen (see 2.2.2.1). This event cancels the recognition process automatically for the automatic recognition mode – see Figure 15.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Recognizer did not detect speech within the period specified in the initialtimeout attribute.</td>
</tr>
<tr>
<td>Default action</td>
<td>Set status = -11</td>
</tr>
</tbody>
</table>

**Event Properties:**

Although the event handler does not receive properties directly, the handler can query the event object for data.

2.2.4.3 onspeechdetected

**onspeechdetected** is fired by the speech recognition platform on the detection of speech. Determining the actual time of firing is left to the platform (which may be configured on certain platforms using the param element, as in 2.2.1.4), so this may be anywhere between simple energy detection (early) or complete phrase or semantic value recognition (late).
This event also triggers `onbargein` on a prompt which is in play (see 2.1.4.2), and may disable the `initialtimeout` of a started `dtmf` object, as described in 2.3.6. This handler can be used in multimodal scenarios, for example, to generate a graphical indication that recognition is occurring, or in voice-only scenarios to enable fine control over other processes underway during recognition.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Recognizer detects speech.</td>
</tr>
<tr>
<td>Default action</td>
<td>Trigger <code>onbargein</code> if prompt is in playback, disable <code>dtmf initialtimeout</code> if started.</td>
</tr>
</tbody>
</table>

**Event Properties:**

Although the event handler does not receive properties directly, the handler can query the event object for data.

2.2.4.4 `onnoreco`

`onnoreco` is a handler for the event thrown by the speech recognition platform when it is unable to return a complete recognition result. The different cases in which this may happen are distinguished by status code. For `listens` in automatic mode, this event stops the recognition process automatically.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Recognizer detects speech but is unable to fully interpret the utterance.</td>
</tr>
<tr>
<td>Default action</td>
<td>Update <code>recoresult</code> and <code>status</code> properties. <code>recoresult</code> may be an empty document or it may hold information provided by the speech recognizer. Status codes are set as follows:</td>
</tr>
</tbody>
</table>

- **status -11**: execution was stopped before speech was detected.
- **status -13**: sound was detected but no speech was able to be interpreted;
- **status -14**: some speech was detected and interpreted but rejected with insufficient confidence (for threshold setting, see the `reject` attribute in 2.2.2.1);
- **status -15**: speech was detected and interpreted, but a complete recognition was unable to be returned between the detection of speech and the duration specified in the `babbletimeout` attribute (see 2.2.2.1).
- **status -30**: (telephony profiles only): `Stop()` invoked by `disconnect`, input not recognized.

**Event Properties:**

Although the event handler does not receive properties directly, the handler can query the event object for data.

2.2.4.5 `onerror`

The `onerror` event is fired if a serious or fatal error occurs with the recognition process (i.e. once the recognition process has been started with a call to the `Start` method). Different types of error are distinguished by status code and are shown in the event object table below.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>The grammar activation or recognition process experiences a serious or fatal problem.</td>
</tr>
<tr>
<td>Default action</td>
<td>Set <code>status</code> property and return null recognition result. The <code>listen's recoresult</code> and <code>text</code> properties are set to empty. Status codes are set as follows:</td>
</tr>
</tbody>
</table>

- **status -1**: A generic (speech) platform error occurred during
Event Properties:
Although the event handler does not receive properties directly, the handler can query the event object for data.

2.2.5 Interaction with DTMF
In telephony profiles which support DTMF input, platforms will implement certain links between listen objects and dtmf objects which simplify the authoring of joint behavior on a single dialog turn. This relationship is discussed in section 2.3.6 of the DTMF input chapter.

2.2.6 Recognition mode
Different scenarios of speech recognition can require subtle differences in behavior from a speech recognizer. Although the starting of the recognition process is standard in all cases – an explicit Start() call from uplevel browsers, or a declarative listen element in downlevel browsers – the means of stopping the process and the return of results may differ.

For example, an end-user using tap-and-talk in a multimodal application may control the period of spoken input to the device by tapping and holding a form field, so the application uses a GUI event (e.g. pen up) to control when recognition will stop and return results. However, in voice-only scenarios such as telephony or hands-free, the user has no direct control over the browser, and the recognition platform must take the responsibility of deciding when to stop recognition and return results (typically once a complete path through the grammar has been recognized). Further, dictation and other scenarios where intermediate results may need to be returned before recognition is stopped not only require an explicit stop but also need to return multiple recognition results to the application before the recognition process is stopped.

Hence the mode attribute on the listen element is used to distinguish the following three modes of recognition: automatic, single and multiple. These are distinguished by how and when the speech recognizer returns results. The return of results is accompanied by the throwing of the onreco event.

SALT defines profiles for the support expected of different modes according to the class of client device in 2.8.1. Generally, automatic mode will be more useful in telephony profiles, single mode in multimodal profiles, and multiple mode in all kinds of dictation scenarios. (It is expected that applications will reflect such profiles in server-side page generation, that is, individual pages will be tailored on a web server to specific classes of client device according to the modality capabilities of that client.)

As noted above, if mode is unspecified, the default recognition mode is ‘automatic’.

Note
Applications may make the assumption that communications between the browser and the recognition platform are ordered correctly in time. This assumption may not always hold true in distributed architectures where heavy loads on the recognition platform cannot guarantee the chronological sequencing of communications across components. For example
A `Stop()` call may be transmitted from browser to platform after the user has stopped speaking, but while the platform is still processing the input. Browser implementations with distributed architectures will clearly need to take this into account.

2.2.6.1 Automatic mode

```xml
<listen mode="automatic" ...
```

![Diagram of Automatic mode listen timeline](image)

Automatic mode is used for recognitions in telephony or hands-free scenarios. The general principle with automatic listens is that the speech platform itself (rather than the application) is in control of when to stop the recognition process. So no explicit `Stop()` call is necessary from the application, because the utterance end will be automatically determined, typically using the `endsilence` value.

Speech detection is signaled by the `onspeechdetected` event. As described in 2.2.4.3, the timing of this event is determined completely by the platform. (If a prompt is in playback when `onspeechdetected` is thrown, the `onbargein`
As soon as a recognition result is available (the \textit{endsilence} time period is used to determine the phrase-end silence which implies recognition is complete), the speech platform automatically stops the recognizer and returns its results. The \texttt{onreco} event is thrown for a successful recognition (i.e. confidence higher than the threshold specified in the \texttt{reject} attribute), and \texttt{onnoreco} for an unsuccessful recognition (i.e. confidence lower than the threshold specified in the \texttt{reject} attribute). This is shown in diagrammatic form in case (1) of Figure 15. Case (2) shows the firing of \texttt{onnoreco} after the \texttt{babbletimeout} period is exceeded, which ends execution of the \texttt{listen}. Case (3) displays an unsuccessful recognition attempt where the recognizer throws \texttt{onnoreco} before the utterance \texttt{endsilence}. Case (4) shows no input from the user, and the resulting throwing of the \texttt{onsilence} event. As noted above, all events except \texttt{onspeechdetected} end the execution of a \texttt{listen} in automatic mode.

2.2.6.2 Single mode

\begin{verbatim}<listen mode="single" \ldots >
\end{verbatim}
Single mode recognitions are typically used for push-to-talk scenarios. In this mode, the return of a recognition result is under the control of an explicit `Stop()` call from the application.

Figure 16 shows the common speech recognition events and their behavior for a single mode `listen`.

Speech detection is signaled by the `onSpeechDetected` event. As described in 2.2.4.3, the timing of this event is determined completely by the platform. (If a prompt is in playback when `onSpeechDetected` is thrown, the `onBargeIn` event will be thrown on the prompt (see 2.1.4.2), and if the prompt’s `bargeIn` attribute is true, playback will first be stopped.)

Case (1) shows the `Stop()` call in action and the possible resulting events of `onReco` or `onnoreco`, according to whether recognition was successful or not. Case (2) illustrates the firing of `onnoreco` in response to the `babbleTimeout`, and this event automatically ends the execution of the `listen`. Case (3) shows how `onnoreco` may
be fired in the case of an unrecognizable utterance, but this does not automatically cease execution of the \texttt{listen}. And case (4) shows how, as with all modes, the \texttt{onsilence} event is thrown if speech is not detected within the timeout period (but for a single mode \texttt{listen} this does not stop recognition). So for single mode listens, the only speech event which automatically halts execution before a stop call is \texttt{onoreco} as a result of \texttt{babbletimeout} (along with the non-speech event \texttt{onerror}).

2.2.6.3 Multiple mode

\begin{verbatim}
<listen mode="multiple" ... >
\end{verbatim}
Multiple mode recognition is useful for "open-microphone" or dictation scenarios. In this mode, recognition results are returned at intervals until the application makes an explicit Stop() call (or the babbletimeout or maxtimeout periods are exceeded). It is important to note that after any onsilence, onreco, or onnoreco event which does not stop
recognition, the maxtimeout and babbletimeout periods are restarted. recoresult is updated by these events as for the other modes of recognition.

For each phrase recognized, an onspeechdetected event is thrown, followed by an onreco event and the return of the phrase result. As with the other modes, onspeechdetected has the effects on prompt playback described in section 2.1.4.2. The decision of what constitutes a valid recognized phrase for the return result is left entirely to the platform. Phrase recognition is shown in case (1) of Figure 17. On the return of the result, the babbletimeout and maxtimeout periods are restarted. Case (2) shows how the exceeding of the babbletimeout results in the onnoreco and the halting of listen. Case (3) displays the throwing of onnoreco in response to unrecognizable input, with the listen object continuing execution and the restarting of the babbletimeout and maxtimeout periods. Case (4) shows the throwing of onsilence in response to no input during the initialtimeout period, and again the execution of the listen object continues.

2.2.7 Events which stop listen execution

The following is a summary of the commands and events that will stop a listen while in execution:

methods
- listen.Stop()
- listen.Cancel()

listen events
- listen.onreco (automatic mode only)
- listen.onnoreco (babbletimeout: all modes)
- listen.onnoreco (unsuccessful recognition: automatic mode only)
- listen.onsilence (automatic mode only)
- listen.onerror

DTMF events (telephony profiles only)
- dtmf.onreco
- dtmf.onnoreco
- dtmf.onsilence
- dtmf.onerror

Recall also that in telephony profiles a hang-up event automatically calls Stop() on an active listen.

2.2.8 Recording with listen

The listen element is also used for recording audio input from the user. Recording may be used in addition to recognition or in place of it, according to the abilities of the platform and its profile. The attributes, properties and methods of listen are used in the recording case with equivalent or appropriate semantics. This section explains these features for recording scenarios, and a full example can be found in section 2.6.8.

For concurrent recognition and recording in a single listen (e.g. using ‘hotword’ recognition to end a recording), both grammar and record may be used. The attributes of record configure the recording process, and the attributes, properties, methods and event handlers of listen should be considered to apply to the speech recognition process. The mode of recognition used will typically determine the overall behavior of such a listen object. The results of both recognition and recording will be contained in the recoresult returned to the browser (whether recognition is a success or a failure) and all the relevant properties of the listen object will be updated.

2.2.8.1 <listen> content for recording

2.2.8.1.1 <record> element

Recording is enabled on a listen element by the use of the record element. Only one record element is permitted in a single listen. The following optional attributes of record are used to configure the recording process:

- **type**: Optional. If unspecified, defaults to G.711 wav file. Formats required by SALT clients which support the basic recording module (see section 2.8.1.3) are G.711 wav (audio/wav: 8kHz 8-bit mono [PCM] single
Speech Application Language Tags (SALT) 1.0

channel) and headerless (audio/basic: 8kHz 8-bit mono [PCM] single channel). (Compression type is expected according to the telephony standard in the country of deployment, e.g. Mu-law in North American deployments, A-law in European deployments, etc.):

- beep optional. Boolean value, if true, the platform will play a beep before recording begins. Defaults to false.

2.2.8.1.2 <grammar> element
If specified in addition to a record element, the grammar element enables speech recognition during the recording process as described in 2.2.1.1. (not all platforms will support this profile). This is useful in certain scenarios, including 'hotword' detection to end recording, or where the audio of recognized input needs to be made available to the application.

2.2.8.1.3 <bind> element
The semantic markup document returned after a recording holds in its root element the following extra attributes relevant to the recording result:

recordlocation: uri of the location of the recorded audio;
recordtype: media-type of the recorded audio.
recordduration: value (in ms) corresponding to the approximate length of the recording;
recordsize: value (in bytes) holding the size of the recorded audio file;

The values of these attributes are copied to the relevant properties of a recording listen object (see 2.2.8.2.2). In the case of a totally unsuccessful recording, recordlocation and recordtype will hold empty strings, and recordduration and recordsize will hold values of zero.

2.2.8.1.4 <param> element
As with typical listens, the param element can be used to specify the platform-specific features of recording, e.g. sampling rate, mu-law/A-law compression, etc.

2.2.8.2 Attributes and properties

2.2.8.2.1 Attributes
The following attributes of listen are used to configure the speech recognizer for recording. For all attributes, where concurrent recognition and recording are performed, the attributes are used as for the speech recognition case in 2.2.2.1.

- initialtimeout: Optional. For recording only, the time in milliseconds between start of recording (if no prompt is in playback) or the end of prompt (if a prompt is in playback) and the detection of speech. This value is passed to the recording platform, and if exceeded, an onsilence event will be thrown from the recognition platform (see 2.2.4.2). If not specified, the speech platform will use a default value.
- babbletimeout: Optional. For recording only, this sets the time limit on the amount of audio that can be recorded once speech has been detected. If babbletimeout is exceeded, the onnoreco event is thrown with status code -15 (see section 2.2.4.4). If babbletimeout is not specified, the speech platform will default to an internal value. A value of 0 effectively disables the timeout.
- maxtimeout: Optional. For recording only, this sets the maximum timeout period in which a recording must be returned, and is used as defined for a typical listen (see 2.2.2.1).
- endsilence: Optional. For recording only, the period of silence in milliseconds after the end of an utterance which must be free of speech after which audio recording is automatically stopped. If unspecified, defaults to a platform internal value.
- reject: Optional. For listens used only for recording, this is ignored.
- xml:lang: Optional. For listens used only for recording, this is ignored.

---

9 audio/wav is widely used as a media type, although it is not formally registered as an rfc. The audio/basic media type is described in http://www.ietf.org/rfc/rfc1521.txt.
10 For recording on a telephony platform this functionality could also be accomplished by most telephony cards. Hence, for recording, the implementation of this feature is left in the hands of platform implementation.
11 Recording platforms may begin writing to file at any time during the initialtimeout period, so the entire length of a recorded file may be anywhere up to the sum of initialtimeout and babbletimeout.
12 This may be interpreted by the platform as an instruction to allow maximum rather than infinite length input.
• **mode**: Optional. For *listens* used only for recording, this is ignored.

### 2.2.8.2.2 Properties
The following properties contain the results returned by the recording process. Those properties which hold values specific to recording obtain the corresponding values from the return document described in 2.2.8.1.3.

- **recoresult** Read-only. As for recognition, the results of recording in an XML DOM node holding the return document described in 2.2.8.1.3. For *listens* used for simultaneous recording and recognition, the return document will hold information for both results.
- **text** Read-only string. (Only used when recognition is enabled along with recording).
- **status** Read-only. Integer holding status code returned by the recognition platform. Status codes from -20 to -24 are relevant for errors specific to the audio recording process (see 2.2.8.4.5). Status code -30 indicates a disconnect in telephony profiles.
- **recordlocation** Read-only. String holding the location of the recorded audio in a URI.
- **recordtype** Read-only. String holding the media type of the recorded audio.
- **recordduration**: Read-only. Integer holding the approximate length of the recording in milliseconds.
- **recordsize**: Read-only. Integer holding the size of the recorded audio file in bytes.

### 2.2.8.3 Object methods
Recording activation can be controlled using the following methods of listen. With these methods, uplevel browsers can start and stop recording, and cancel recordings in progress.

**2.2.8.3.1 Start**
This method is used to start audio recording.

**Syntax:**
```
Object.Start()
```

**Parameters:**
None.

**Return value:**
None.

**Exception:**
The method sets a non-zero status code and fires an onerror event if it fails. See the onerror event description in section 2.2.4.5 for the non-zero status codes.

**2.2.8.3.2 Stop**
This method is used to stop audio recording. For recording-only *listens*, unless there is a recording error, the onreco event will fire as a result, once the platform completes the audio recording. In telephony profiles, the `Stop()` method is executed automatically by the detection of a disconnect (the order of firing is as follows: listen, dtmf, PromptQueue), which results in onreco and a status code of -30.

**Syntax:**
```
Object.Stop()
```

**Parameters:**
None.

**Return value:**
None.

**Exception:**
There are no explicit exceptions associated with this method. However, an onerror event may be fired in the case of failure and the status codes as outlined in section 2.2.8.4.5 are set.

**2.2.8.3.3 Cancel**
This method is used to cancel audio recording in a recording only *listen*. Any written audio data may be removed by the platform. No events are fired when this method is called.

**Syntax:**
```
Object.Cancel()
```
2.2.8.3.4 Activate
This method is used only when recognition is enabled, and is as described in section 2.2.3.4.

2.2.8.3.5 Deactivate
This method is used only when recognition is enabled, and is as described in section 2.2.3.5.

2.2.8.4 Recording events
A recording listen supports the following events, whose handlers may be specified as attributes of the listen element.

For listens which execute recording only (without recognition), the event behavior is as for a listen of automatic mode (see section 2.2.6.1.). For listens which accomplish recognition along with recording, the mode of recognition used will determine which events are thrown and their behavior (see section 2.2.6 for different modes of recognition).

2.2.8.4.1 onspeechdetected
onspeechdetected is fired by the speech recognition platform on the detection of speech. Determining the actual time of firing is left to the platform (which may be configured on certain platforms using the param element, as in 2.2.1.4), so this may be anywhere between simple energy detection (early) or more sophisticated speech detection (late). This event also triggers onbargein on a prompt which is in play (see 2.1.4.2), and may disable the initialtimeout of a started dtmf object, as described in 2.3.6.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Speech is detected.</td>
</tr>
<tr>
<td>Default action</td>
<td>Trigger onbargein if prompt is in playback, disable dtmf initialtimeout if started.</td>
</tr>
</tbody>
</table>

**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

2.2.8.4.2 onreco
For recording-only listens, this event is fired when audio recording has completed. The recoresult property is returned with the recording result and properties are updated according to the previous sections.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Recording is accomplished</td>
</tr>
</tbody>
</table>

**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data (see the use of the event object in the example below).

2.2.8.4.3 onsilence
For recording-only listens, this event is fired when no speech is detected by the platform before the duration of time specified in the initialtimeout attribute on the listen (see 2.2.8.2.1). This event cancels the audio recording process automatically.
Event Object Information:

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Recognizer did not detect speech within the period specified in the initialtimeout attribute.</td>
</tr>
<tr>
<td>Default action</td>
<td>Set status = -11</td>
</tr>
</tbody>
</table>

Event Properties:

Although the event handler does not receive properties directly, the handler can query the event object for data.

2.2.8.4 onnoreco

For recording-only listens, this event is thrown when the babbletimeout period on the recording has been exceeded. This is a common occurrence in voice mail scenarios when the time allotted for leaving a message is exceeded by the user. The platform also returns the recording results via the recoresult property, and applications will typically apply the same handler as for onreco.

Event Object Information:

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>babbletimeout expires during audio recording.</td>
</tr>
</tbody>
</table>
| Default action | Set status property and return recording result in recoresult. Status codes are set as follows:

| status -15: speech was detected and recording made but babbletimeout was exceeded (see 2.2.2.1). |

Event Properties:

Although the event handler does not receive properties directly, the handler can query the event object for data.

2.2.8.4.5 onerror

The onerror event is fired if a serious or fatal error occurs with the recording process. Different types of error are distinguished by status code and are shown in the event object information table below.

Event Object Information:

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>The recording process (once the Start method has been invoked) experiences a serious or fatal problem.</td>
</tr>
</tbody>
</table>
| Default action | Set status property and return empty recording result. Recording status codes are set as follows:

| status -20: Failure to record file locally on the platform |
| status -21: Unsupported codec |
| status -22: Unsupported format (if neither format nor codec are unsupported, only one of the values need be set) |
| status -23: Error occurred during streaming to a remote server. |
| status -24: An illegal property/attribute setting that causes a problem with the recording request. |
| status -30: Recording was attempted after a disconnect (telephony profiles). |

Event Properties:

Although the event handler does not receive properties directly, the handler can query the event object for data.

2.2.8.5 Timeline for recording listen

The following figure, Figure 18, shows the typical events of a listen which is executing audio recording (without concurrent recognition).
Speech detection is signaled by the `onSpeechDetected` event. As described in 2.2.4.3, the timing of this event is determined by the platform. (If a prompt is in playback when `onSpeechDetected` is thrown, the `onBargein` event will be thrown on the prompt (see 2.1.4.2), and if the prompt's `bargein` attribute is true, playback will first be stopped.)

As soon as a recording is available (the `endSilence` time period can be used to determine the silence which implies that the recording is over), the speech platform automatically stops the recording and returns results. The `onReco` event is thrown, as shown in diagrammatic form in case (1). Case (2) shows the firing of `onReco` after the `babbleTimeout` period is exceeded, which ends execution of the `listen` and returns the recording. Case (3) shows an explicit `Stop()` call which returns the recording result and throws the `onReco` event (recall that a disconnect in telephony profiles automatically calls the `Stop()` method). Case (4) shows no input from the user, and the consequent throwing of the `onSilence` event.

Figure 18: Recording listen timeline
2.2.8.6 Stopping audio recording

Audio recording is stopped by any of the means of stopping a \texttt{listen} object in automatic mode (see section 2.2.7). Since a result is always returned to the \texttt{listen} object, a recording \texttt{listen} which is stopped by DTMF input or a hang-up (as in many voice mail applications, for instance) will always contain a recording result if available.

The following is a summary of the commands and events that will stop a record-only \texttt{listen} while in execution:

methods
- \texttt{listen.Stop()}
- \texttt{listen.Cancel()}

\texttt{listen} events
- \texttt{listen.onreco}
- \texttt{listen.onnoreco}
- \texttt{listen.onsilence}
- \texttt{listen.onerror}

DTMF events (telephony profiles only)
- \texttt{dtmf.onreco}
- \texttt{dtmf.onnoreco}
- \texttt{dtmf.onsilence}
- \texttt{dtmf.onerror}

Recall also that in telephony profiles a hang-up event automatically calls \texttt{Stop()} on an active \texttt{listen}.

2.2.9 Advanced speech recognition technology

It should be clear that advanced speech recognition technologies such as speaker verification or enrollment are enabled by the \texttt{listen} element as it is currently defined in SALT, although optimal methods for accomplishing such mechanisms may not be portable across platforms.

2.3 DTMF input: <\texttt{dtmf}>

The \texttt{dtmf} element is used in telephony applications to specify possible DTMF inputs and a means of dealing with the collected results and other DTMF events. Like \texttt{listen}, its main elements are \texttt{grammar} and \texttt{bind}, and it holds resources for configuring the DTMF collection process and handling DTMF platform and collection events.

2.3.1 \texttt{dtmf} content

Mirroring the \texttt{listen} recognition element, the \texttt{dtmf} element holds as content the \texttt{grammar} and \texttt{bind} elements, and may also be configured in extensible ways with the \texttt{param} element.

2.3.1.1 <\texttt{grammar}>

This is a grammar, as defined in section 2.2.1.1. The only difference between a speech grammar and a DTMF grammar is that the DTMF grammar will hold DTMF keys as tokens, rather than words of a particular language. So for a DTMF grammar, the \texttt{xml:lang} attribute is not meaningful, and within the grammar itself, terminal rules will contain as possible tokens only the digits 0-9, *, # and A, B, C and D. In all other respects, the \texttt{grammar} element is identical to the speech recognition \texttt{grammar} element in section 2.2.1.1.

2.3.1.2 <\texttt{bind}>

The \texttt{bind} element is a declarative way to assign the DTMF result to a field in the host page, and is defined in section 2.2.1.2. \texttt{bind} acts on the XML in the result returned by DTMF collection in exactly the same way as it does for \texttt{listen}.

The following example demonstrates how to allow consecutive DTMF input into multiple fields, using DTMF grammars and \texttt{bind} to update the fields.

```html
  ...
  <input type="text" name="iptAreaCode" onFocus="dtmfAreaCode.start()" />
</html>
```
2.3.1.3 DTMF configuration: <param>

Additional, non-standard configuration of the DTMF engine is accomplished with the use of the param element which passes parameters and their values to the platform. param is a child element of dtmf.

The exact nature of the configurative parameters will differ according to the proprietary platform used. Values of parameters may be specified in an XML namespace, in order to allow complex or structured values.

**param element**

param: Optional. Used to pass parameter settings to the speech platform.

param content

Attributes:

- **name**: required. The name of the parameter to be configured.
- **xmlns**: optional. Specifies a namespace and potentially a schema for XML content of the parameter.

So, for example, the following syntax:

```html
<salt:param name="myDTMFParam"> myDTMFValue </salt:param>
```

could be used to specify a parameterization on particular DTMF platform.

Note that in HTML profiles, page-level parameter settings may also be defined using the meta element (see 2.8.2.2.1.5).

2.3.2 dtmf attributes and properties

2.3.2.1 Attributes

- **id**: optional. The identifier of the dtmf element. Must be a valid XML Name and unique within the document (i.e. of XML type ID).
- **initialtimeout**: Optional. The time in milliseconds between start of collection and the first key pressed. If exceeded, the onsilence event is thrown and status property set to -11. A value of 0 effectively disables the timeout. If the attribute is not specified, the speech platform will use a default value.
- **interdigittimeout**: Optional. Timeout period for adjacent DTMF keystrokes, in milliseconds. A value of 0 effectively disables the timeout. If unspecified, defaults to the telephony platform’s internal setting. When exceeded, the platform throws an onnoreco event and sets the status property to -16.
- **endsilence**: optional. The timeout period when input matches a complete path through the grammar but further input is still possible. This timeout specifies the period of time in which further input is permitted after the complete match. Once exceeded, onreco is thrown. (For a complete grammar match where further input is not possible, the endsilence period is not required, and onreco is thrown immediately.) If this attribute is not supported directly by a platform, or unspecified in the application, the value of endsilence defaults to that used for interdigittimeout.
• **preflush**: Optional. Boolean flag indicating whether to automatically flush the DTMF buffer on the underlying telephony interface card before activation. If unspecified, defaults to false (in order to facilitate type-ahead applications).

### 2.3.2.2 Properties

- **dtmfresult**: Read only. XML node holding the DTMF result. This is updated at the end of DTMF collection, and holds an XML document containing semantic markup language. Semantic markup language is discussed in section 2.2.1.2.
- **text**: Read-only string containing tokens of the actual keys pressed during recognition. This string is appended with every key press event (in-grammar or out-of-grammar) received by the dtmf object, so it is updated on every onkeypress, onreco and onnoreco event. The string does not hold white space between tokens.
- **status**: Read-only. Integer holding the status code returned by DTMF collection. The status property is only meaningful after status-setting events are thrown by the dtmf object, and applications should examine it in the handler of the relevant event. Possible values are 0 for successful collection, the failure values -1 to -9 (as defined in the exceptions possible on the Start method (section 2.3.3.1)), status -11 on the reception of onsilence (see 2.3.4.4), and status -13 or -16 on an onnoreco (see 2.3.4.3). (These values reflect corresponding status codes in the listen object of section 2.2.2.2.) Status -30 indicates a telephony disconnect event.

### 2.3.3 dtmf methods

DTMF collection may be controlled using the following methods on the dtmf object. With these methods, browsers can start, stop and flush dtmf objects.

#### 2.3.3.1 Start

The **Start** method starts the DTMF collection process, using the grammars defined within the object.

**Syntax:**

```
Object.Start();
```

**Parameters:**

None.

**Return value:**

None.

**Exception:**

The method sets a non-zero status code and fires an onerror event if it fails. See the onerror event description in section 2.3.4.5 for the non-zero status codes.

Only one dtmf object may be started at a given time. If **Start()** is called on a dtmf object which is already in execution, the call has no effect. If **Start()** is called on a dtmf object while another is in execution, the onerror event is thrown on the object on which the second **Start()** was attempted.

#### 2.3.3.2 Stop

Stop dtmf collection and return results received up to the point when collection was stopped. (Any subsequent keystrokes entered by the user, however, will remain in the platform buffer unless explicitly flushed.) The result of calling **Stop()** will be an onreco or onnoreco event and the return of a result, or an onerror event. If the dtmf object has not been started, this call has no effect. In telephony profiles, the **Stop()** method is executed automatically by the detection of a disconnect (the order of firing is as follows: listen, dtmf, PromptQueue), and the relevant event thrown with status code -30.

**Syntax:**

```
Object.Stop();
```

**Parameters:**

None.

**Return value:**

None.

**Exception:**

There are no explicit exceptions associated with this method. However, if there is any problem an onerror event is fired and the status codes as outlined in section 2.3.4.5 are set.
2.3.3 Flush
Flush the DTMF buffer. Flush has no effect if called while the dtmf object is started.

**Syntax:**

```javascript
Object.Flush();
```

**Parameters:**
None.

**Return value:**
None

**Exception:**
There are no explicit exceptions associated with this method. However, if there is any problem an onerror event is fired and the status codes as outlined in section 2.3.4.5 are set.

2.3.4 dtmf events
When returning a result from DTMF collection, browsers will update the recoresult property for both successful and unsuccessful DTMF recognitions. However, applications should assume this property holds a valid result only in the case of successful recognitions. In the case of unsuccessful or aborted recognitions, the result may be an empty document (or it may hold extra information which applications are free to use or ignore).

2.3.4.1 onkeypress
Fires on every pressing of a DTMF key which is legal according to the input grammar (a key which is out-of-grammar triggers an onnoreco event). Immediately prior to the event firing, the token corresponding to the key pressed is appended to the value of the text property of the dtmf element. If a prompt is in playback, the onkeypress event will trigger the onbargein event on the prompt (and cease its playback if the prompt's bargein attribute is set to true). If a listen element is active, the first onkeypress event has the effect described in section 2.3.6.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Key press on the DTMF key pad.</td>
</tr>
<tr>
<td>Default action</td>
<td>Appends text property with key pressed</td>
</tr>
</tbody>
</table>

**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

2.3.4.2 onreco
Fires when a DTMF recognition is complete. This event stops the current dtmf object automatically and updates dtmfresult with the results of recognition. If a listen is also active, this event stops the listen object, as described in section 2.3.6. The onreco handler is typically used for programmatic analysis of the recognition result and processing of the result into fields on the page.

DTMF recognition is considered complete and onreco fired in the following circumstances:

1. Immediately after the input sequence matches a complete path through the grammar and further input is not possible according to that grammar.
2. After the period specified in the endsilence attribute in the case where the input sequence matches a complete path through the grammar but further input is still possible according to that grammar. (So setting an endsilence period of zero would fire onreco immediately a complete path through the grammar is matched, and have the same behavior as 1.)

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>DTMF recognition is complete.</td>
</tr>
<tr>
<td>Default action</td>
<td>Returns result in dtmfresult. Set status codes as follows:</td>
</tr>
</tbody>
</table>

---

13 For HTML and XHTML, this overrides the default onkeypress event inherited from the HTML control. Only DTMF keypresses fire this event.
**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

### 2.3.4.3 onnoreco
Fires when a key is pressed which is not legal according to the DTMF grammar, or when `interdigittimeout` is exceeded (or on the `Stop()` call when input is not a complete grammar match). This event stops the DTMF object automatically, appends the `text` property with key pressed (if an illegal key was pressed) and updates `dtmfresult` with a result (this may be an empty document or it may hold the out-of-grammar input). If a listen is also active, this event stops the `listen` object, as described in section 2.3.6.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Illegal key press, or exceeding of <code>interdigittimeout</code> period when input is incomplete.</td>
</tr>
<tr>
<td>Default action</td>
<td>Stops DTMF collection. Appends <code>text</code> property with key pressed (if applicable), updates <code>dtmfresult</code> and sets <code>status</code> property. Status codes are set as follows:</td>
</tr>
<tr>
<td>status -13</td>
<td>out-of-grammar DTMF keypress.</td>
</tr>
<tr>
<td>status -16</td>
<td><code>interdigittimeout</code> was exceeded.</td>
</tr>
<tr>
<td>status -30</td>
<td>a disconnect invoked the <code>Stop()</code> method.</td>
</tr>
</tbody>
</table>

**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

### 2.3.4.4 onsilence
`onsilence` handles the event of no DTMF collected by the platform before the duration of time specified in the `initialtimeout` attribute (see 2.3.2.1). This event stops the `dtmf` object automatically.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>No DTMF input detected within the period specified in the <code>initialtimeout</code> attribute.</td>
</tr>
<tr>
<td>Default action</td>
<td>Set status = -11</td>
</tr>
</tbody>
</table>

**Event Properties:**
Although the event handler does not receive properties directly, the handler can query the event object for data.

### 2.3.4.5 onerror
The `onerror` event is fired if a serious or fatal error occurs with the DTMF collection process. Different types of error are distinguished by status code and are shown in the event object information table below.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>The DTMF collection process experiences a serious or fatal problem.</td>
</tr>
<tr>
<td>Default action</td>
<td>On encountering an error, the <code>dtmf</code> object is stopped and status codes are set as follows:</td>
</tr>
<tr>
<td>status -1</td>
<td>A generic platform error occurred during DTMF collection.</td>
</tr>
<tr>
<td>status -3</td>
<td>An illegal property/attribute setting that causes a</td>
</tr>
</tbody>
</table>

---

14 A status code of -30 for the `dtmf` object’s `onreco` event is only possible if the disconnect event occurred after the input sequence matched a valid path through the grammar, but before the endsilence period expires.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to find resource</td>
<td>-4</td>
<td>in the case of DTMF this is likely to be the URI of a DTMF grammar.</td>
</tr>
<tr>
<td>Failure to load or compile a grammar resource</td>
<td>-5</td>
<td></td>
</tr>
<tr>
<td>Collection was attempted without active grammars</td>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>Collection was attempted while another dtmf object</td>
<td>-9</td>
<td>was in execution.</td>
</tr>
<tr>
<td>DTMF collection attempted after a disconnect.</td>
<td>-30</td>
<td></td>
</tr>
</tbody>
</table>

**Event Properties:**

Although the event handler does not receive properties directly, the handler can query the event object for data.

### 2.3.5 DTMF event timeline

The following diagram illustrates typical event possibilities for the `dtmf` element.
Case (1) shows a successful DTMF recognition. `onkeypress` events are fired for every in-grammar keypress, and `onreco` is thrown either when a complete grammar match has been made or after the `endsilence` period if further input was permitted in the grammar. Case (2) shows the `onnoreco` event being thrown by the exceeding of the `interdigittimeout` period. Case (3) shows that an out-of-grammar keypress fires `onnoreco`. Case (4) shows `onsilence` being thrown if no input is entered before the `initialtimeout` period elapses. As noted, all events except `onkeypress` end DTMF collection.

2.3.6 Using listen and dtmf simultaneously

Many telephony applications typically permit speech and/or DTMF to be used at the same time. To simplify the authoring of such scenarios, SALT platforms implement a model of default behavior whereby detection and successful recognition of one mode of input need not interfere with the other. In general, this means that the application only has to worry about receiving a single recognition result, or other event, even when both objects are started. For finer level behavior, `listen` or `dtmf` events can be handled individually without affecting the other mode.
SALT enables this in two ways: (i) the disabling of initial timeouts on the other mode on detection of input, and (ii) the automatic cancellation of one mode when the other mode comes to an end. This behavior is discussed in the following two subsections, and the section completes with a diagram illustrating the interplay of the objects.

2.3.6.1 Disabling timeouts

Once the platform has detected input in one mode, it disables the initialtimeout of the other mode. That is, when the initialtimeout period is exceeded, the object is stopped automatically but its event is not thrown to the handler. This is effectively an acknowledgement that user input is occurring, and therefore the calling of an onsilence handler on the other, unused mode is irrelevant. This prevents, for example, a listen's onsilence from calling its handler while the user is entering DTMF. The manifestation of detection is a keypress (onkeypress event) for dtmf, and the onspeechdetected event for listen\(^{15}\).

\[
\begin{align*}
dtmf.onkeypress & \quad \rightarrow \text{disable listen timeouts} \\
listen.onspeechdetected & \quad \rightarrow \text{disable dtmf timeouts}
\end{align*}
\]

With initialtimeout disabled, such 'unused' objects do not throw an onsilence event. If the timeout of the unused mode does not expire, both objects remain active until otherwise stopped. This may be useful, for example, in scenarios such as where DTMF keypresses are used to control playback of the prompt, while voice commands effect dialog navigation (e.g. in an e-mail reader application). If the application author wishes to stop the unused object on detection of the other input mode, this is possible by adding such a Stop command to the relevant event handler of the 'used' mode.

Once disabled, the initialtimeout is not re-enabled or re-started. That is, once the platform detects one mode of input, onsilence will never be thrown on either mode. This should never be a problem, since other timeouts are still active on any 'used' modes (endsilence, babbletimeout and interdigittimeout), so they will always eventually stop.

2.3.6.2 Automatic stop

When one mode stops and throws an event of onsilence, onnoreco, onreco or onerror, the other mode is automatically stopped\(^{16}\). (Stopping actually occurs before the object receives the event, in order for the event handler functions to operate under a predictable situation.) A result will be returned in the relevant property of the automatically stopped object (recoresult for listen, dtmfresult for dtmf) and the status property may be given a particular code.

\[
\begin{align*}
dtmf.onsilence & \quad \rightarrow \text{stop listen} \\
dtmf.onnoreco & \quad \rightarrow \text{stop listen} \\
dtmf.onreco & \quad \rightarrow \text{stop listen} \\
dtmf.onerror & \quad \rightarrow \text{stop listen} \\
listen.onsilence & \quad \rightarrow \text{stop dtmf} \\
listen.onnoreco & \quad \rightarrow \text{stop dtmf} \\
listen.onreco & \quad \rightarrow \text{stop dtmf} \\
listen.onerror & \quad \rightarrow \text{stop dtmf}
\end{align*}
\]

This means that such events from either mode which signal the end of a dialog turn do not need to be caught twice. So the firing of onsilence will be thrown only to the started listen or to the started dtmf object, but not to both. Similarly, the other mode is stopped automatically on (i) a misrecognition or out-of-grammar DTMF sequence (listen.onnoreco or dtmf.onnoreco); or (ii) a successful recognition (listen.onreco, dtmf.onreco).

This allows the application author to write modular code for these handlers which does not need to take explicit account of which objects have been started. And since a result is returned for the automatically stopped object, it allows scenarios where one mode is actually used to force a result return of the other, for example using dtmf to stop audio recording.

\(^{15}\) Recall from section 2.2.4.3 that the decision when to throw onspeechdetected is left to the platform - this permits platforms to operate robust mechanisms whereby throwing the event later - i.e. at a safer time - will not unnecessarily disable the dtmf timeout.

\(^{16}\) Of course this does not apply to a mode which is stopped explicitly by the stopping of the other mode (as described in this paragraph).
2.3.6.3 listen and dtmf interaction event timeline

The model described above is illustrated in the following event diagram, which shows possible interactions between the two started modes of input.

![Figure 20: listen and dtmf event interaction](image)

2.3.7 Events which stop dtmf execution

The following is a summary of the commands and events that will stop dtmf while in execution:

methods
- dtmf.Stop()
dtmf events
- dtmf.onreco
- dtmf.onnoreco
- dtmf.onsilence
- dtmf.onerror

listen events
- listen.onreco
- listen.onnoreco
- listen.onsilence

Recall also that a telephony hang-up event automatically calls Stop() on an active dtmf object.

2.4 Platform messaging: <smex>

`smex` short for Simple Messaging EXtension, is a SALT element that communicates with the external component of the SALT platform. It can be used to implement any application control of platform functionality such as logging and telephony control. As such, `smex` represents a useful mechanism for extensibility in SALT, since it allows any new functionality to be added through this messaging layer.

On its instantiation, the object is directed to establish an asynchronous message exchange channel with a platform component through its configuration parameters (specified in `param` elements) or attributes. The `smex` object can send or receive messages through this channel. The content of a message to be sent is defined in the `sent` property. Whenever the value of this property is updated (either on page load or by dynamic assignment through script or binding), the message is sent to the platform. The `smex` element can also receive XML messages from the platform component in its `received` property. The `onreceive` event is fired whenever a platform message is received. Since the `smex` object's basic operations are asynchronous, it also maintains a built-in timer for the manipulation of timeout settings. `ontimeout` and `onerror` events may also be thrown.

The `smex` object makes no requirement on the means of communication with platform components. It should also be noted that the `smex` object has the same life span as other XML elements, that is, it will be destroyed when its hosting document is unloaded. While in many cases, the `smex` object can perform automatic clean-up and release communication resources when it is unloaded, there might be use cases (e.g., call control) in which a persistent communication link is desirable across pages. For those cases, SALT places the responsibility of relinquishing the allocated resources (e.g., closing the socket) on the application.

The `smex` object also is neutral on the format (schema) of messages. In order to encourage interoperability, however, the conformance criteria in Part 4 recommend that implementations support a known schema for common functionality, with a strong preference for existing standard message formats. Such a schema for telephony call control is suggested in section 2.4.4, and 4.3. In essence, SALT allows both platform and application developers to take the full advantage of the standardized extensibility of XML to introduce innovative and perhaps proprietary features without necessarily losing interoperability.

2.4.1 `smex` content

`smex` may have the following child elements:

2.4.1.1 bind
This is the same element as described in section 2.2.1.2. It operates on the XML document contained in the message received by the browser, so the XPath query held in the `value` attribute will match an XML pattern in this document.

2.4.1.2 param
`param` is used to provide platform-specific parameters for the `smex` object. Each `param` element may be named using a `name` attribute, with the contents of the `param` element being the value of the parameter.

The exact nature of the configurative parameters will differ according to the proprietary platform used. Values of parameters may be specified in an XML namespace, in order to allow complex or structured values.
**param element**

**param:** Optional. Used to pass parameter settings to the speech platform.

**param content**

Attributes:
- **name:** required. The name of the parameter to be configured.
- **xmlns:** optional. Specifies a namespace and potentially a schema for XML content of the parameter.

So, for example, the following syntax:

```xml
<param name="myPlatformParam">myParamValue</param>
```

could be used to specify a parameterization of the message interface to the platform.

Note that in HTML profiles, page-level parameter settings may also be defined using the **meta** element (see 2.8.2.2.1.5).

### 2.4.2 smex attributes and properties

#### 2.4.2.1 smex attributes

The **smex** object has the following attributes:

- **id:** optional. The identifier of the **smex** element. Must be a valid XML Name and unique within the document (i.e. of XML type ID).
- **sent:** Optional. String corresponding to the message to be sent to the platform component. Whenever a non-null value is assigned to this attribute, its contents are dispatched.
- **timer:** Optional. Number in milliseconds indicating the time span before a timeout event will be triggered. The clock starts ticking when the property is assigned a positive value (this may be on document load, if the attribute is specified declaratively). The value can be changed when a countdown is in progress. A zero or negative value stops the clock without triggering the timeout event. The default is 0, meaning no timeout.

#### 2.4.2.2 smex properties

In addition to the attributes, the **smex** element holds the following properties:

- **received:** Read-only. XML DOM Node data indicating the received message. The message is held as the value of this property until the next **onreceive** event is ready to fire.
- **status:** Read-only. Integer indicating the recent status of the object. The possible values are 0, -1, and -2, which indicate, respectively, normal, timeout expired, and communication with the platform cannot be established or has been interrupted. Platform-specific error messages are conveyed through the **received** property. For the cases that the error message is successfully delivered, the status code is 0.

### 2.4.3 smex events

The **smex** object has the following events:

#### 2.4.3.1 onreceive

The **onreceive** event is fired when the browser receives a platform message. If there are any directives declared by the **bind** elements, those directives will first be evaluated before the event is fired. Prior to the firing, the **received** property is updated with the message content.

**Event Object Information:**

<table>
<thead>
<tr>
<th>Bubbles</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>To invoke</td>
<td>Platform message received by the browser.</td>
</tr>
<tr>
<td>Default action</td>
<td><strong>bind</strong> directives are evaluated, the <strong>received</strong> property is updated, and <strong>status</strong> code set to zero.</td>
</tr>
</tbody>
</table>

**Event Properties:**

Although the event handler does not receive properties directly, the handler can query the event object for data.
2.4.3.2 ontimeout
The ontimeout event is fired when the timeout expires.

<table>
<thead>
<tr>
<th>Event Object Information:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubbles</td>
</tr>
<tr>
<td>To invoke</td>
</tr>
<tr>
<td>Default action</td>
</tr>
</tbody>
</table>

Event Properties:
Although the event handler does not receive properties directly, the handler can query the event object for data.

2.4.3.3 onerror
The onerror event is fired when a communication error is encountered. When the event fires, the status property is updated with a corresponding error code as described above.

<table>
<thead>
<tr>
<th>Event Object Information:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubbles</td>
</tr>
<tr>
<td>To invoke</td>
</tr>
<tr>
<td>Default action</td>
</tr>
</tbody>
</table>

Event Properties:
Although the event handler does not receive properties directly, the handler can query the event object for data.

2.4.4 Using smex for telephony call control
Browsers which support the use of smex messages for implementing call control functionality on a telephony platform are recommended to support the ECMA 323 standard XML Protocol for Computer Supported Telecommunications Applications (CSTA) Phase III, as specified at [http://www.ecma.ch/ecma1/STAND/ecma-323.htm](http://www.ecma.ch/ecma1/STAND/ecma-323.htm), which specifies an XML protocol for the CSTA services described in ECMA 269. (Note also the alternative CallControl object specified in Part 3).

2.5 Logging
This chapter specifies the basic requirements for diagnostic logging, and how uplevel SALT browsers implement these requirements.

2.5.1 Overview
Uplevel browsers which support logging provide basic diagnostic logging functionality through a global script method called LogMessage(). This is used to write output to a location of the platform’s choosing.

This function allows the application to log to the same location as other component objects. For example, the listen, dtmf, and prompt elements, and CallControl and other objects may all send log messages to an Operation, Administration and Maintenance (OA & M) server – in concurrence or separately. In order to enable operations administrators to monitor a system through a single, centralized point, applications should be able to send messages to the same location at any time.

2.5.2 Format
The format of the script function is:

```
LogMessage(id, message); where id and message are both strings.
```

- **id** will typically be used to specify the "class" of the message for filtration purposes; for example, an application may define SYSTEM_CRITICAL and use this string to filter all fatal errors to a specific location. This is also useful in off-line analysis and reporting.
- **message** is the content of the diagnostic message itself.

2.5.3 Requirements
- The platform is responsible for providing the global script method. To ensure portability among all SALT platforms, no additional (mandatory) parameters must be added to the logging function. The platform must be
able to handle the `LogMessage()` function under any circumstance. With respect to observing the semantics of the function, platforms are free to treat the function in the manner of their own choosing. Possible actions may be:

- write it to a timestamped log file
- write it to standard out (a console, for example)
- ignore it (for example, send it to `/dev/null` or equivalent).

Section 9.5.1 illustrates how a platform implementing `smex` can use the `smex` element to implement the requirements specified above for the logging function.

## 2.6 SALT illustrative examples

### 2.6.1 Controlling dialog flow

#### 2.6.1.1 Click to talk

This simple example shows how, in a multimodal application, GUI events can be wired to SALT commands such as beginning a recognition turn. In this example, pressing the button named `buttonCityListen` starts the listen named `listenCity`, which holds a grammar of city names, and a `bind` command to transfer the value into the input control named `txtBoxCity`.

```html
<!-- HTML -->
  <form id="travelForm">
    <input name="txtBoxCity" type="text" />
    <input name="buttonCityListen" type="button" onClick="listenCity.Start();" />
  </form>
<!-- SALT -->
<div>
  <salt:listen id="listenCity">
    <salt:grammar name="g_city" src="./city.grxml" />
    <salt:bind targetelement="txtBoxCity" value="/city[1]" />
  </salt:listen>
</html>
```

#### 2.6.1.2 Dialog flow with HTML and scripting

The following examples show how scripting may be used in HTML environments to control dialog flow. The two examples show how the developer can precisely script this flow, since full control of activation of the speech interface is available through algorithms in the script (the `RunAsk` and `RunSpeech` functions respectively).

#### 2.6.1.2.1 Form-filling

This example is a fuller version of the dialog shown in the Scenarios section of the Introduction (1.2), which introduces handlers for the case of misrecognitions.

```html
<!-- HTML -->
  <body onload="RunAsk();">
    <form id="travelForm">
      <input name="txtBoxOriginCity" type="text" />
      <input name="txtBoxDestCity" type="text" />
    </form>
    <salt:prompt id="askOriginCity" >Where would you like to leave from? </salt:prompt> 
    <salt:prompt id="askDestCity" >Where would you like to go to? </salt:prompt> 
    <salt:prompt id="sayDidntUnderstand" oncomplete="runAsk()" >Sorry, I didn't understand. </salt:prompt> 
    <salt:listen id="recoOriginCity" onreco="procOriginCity()" onnoreco="sayDidntUnderstand.Start()" >
```

© SALT Forum founding companies  page 60
2.6.1.2.2 Form-filling and giving help

This example shows how to implement a simple dialog flow which seeks values for input boxes and offers context-sensitive help for the input. It uses the title attribute on the HTML input mechanisms (used in a visual browser as a "tooltip" mechanism) to help form the content of the help prompt.

```html
  <title>Context Sensitive Help</title>
  <head>
    <script>
      <![CDATA[
        var focus;
        function RunSpeech() {
          if (trade.stock.value == "") {
            focus="trade.stock";
            p_stock.Start();
            return;
          }
          if (trade.op.value == "") {
            focus="trade.op";
            p_op.Start();
            return;
          }
          //.. repeat above for all fields
          trade.submit();
        }
        function handle() {
          res = event.srcElement.recoresult;
          if (res.value == "help") {
            text = "Please just say ";
            text += document.all[focus].title;
            p_help.Start(text);
          }
        }
      ]]>
    </script>
  </head>
</html>
```
2.6.1.3 Downlevel dialog flow

This example asks and confirms with the caller for a London football team without using script. It demonstrates a system initiative dialog. However, since the data and the user interface markup are separate, application developers only need to change the speech section when changing interaction style to mixed initiative. The data section remains the same.

The example works in the following way: when an incoming call comes in, the bind in smex starts the welcoming prompt and the corresponding listen object. Depending on the recognition results, the bind directives in the listen object guide the execution using declarative logic. Finally, when the value is confirmed, the form is submitted. All of this is achieved without a single line of script.

</body>
<form id="get_team">
  <input name="team" />
  <input name="uid" type="hidden"/>
</form>

<!-- The speech section -->
<salt:prompt id="welcome">
  Welcome, caller!
</salt:prompt>
<salt:prompt id="ask">
  Which team would you like the latest results for: Arsenal, Chelsea, Spurs or West Ham?
</salt:prompt>
<salt:prompt id="confirm">
  I heard <value targetelement="team" />. Is this correct?
</salt:prompt>
<salt:prompt id="thanks">
  Thank you. Please wait while I get the latest results.
</salt:prompt>
<salt:prompt id="retry">
  Okay, let's do this again
</salt:prompt>
<salt:prompt id="reprompt">
  Sorry, I missed that.
</salt:prompt>

<salt:listen id="listen_team">
  <salt:grammar src="./teamtypes.grxml" />
  <salt:bind test="/[@confidence $gt$ 10]"
    targetelement="team" value="/team" />
  <salt:bind test="/[@confidence $gt$ 10]"
    targetelement="confirm" targetmethod="start" />
  <salt:bind test="/[@confidence $gt$ 10]"
    targetelement="listen_yesno" targetmethod="start" />
  <salt:bind test="/[@confidence $le$ 10]"
    targetelement="reprompt" targetmethod="start" />
  <salt:bind test="/[@confidence $le$ 10]"
    targetelement="ask" targetmethod="start" />
  <salt:bind test="/[@confidence $le$ 10]"
    targetelement="listen_team" targetmethod="start" />
</salt:listen>

<salt:listen id="listen_yesno">
  <salt:grammar src="./yesno.grxml" />
  <salt:bind test="/yes[@confidence $gt$ 10]"
    targetelement="thanks" targetmethod="start" />
  <salt:bind test="/yes[@confidence $gt$ 10]"
    targetelement="get_team" targetmethod="submit" />
  <salt:bind test="/no or ./[@confidence $le$ 10]"
    targetelement="retry" targetmethod="start" />
  <salt:bind test="/no or ./[@confidence $le$ 10]"
    targetelement="ask" targetmethod="start" />
  <salt:bind test="/no or ./[@confidence $le$ 10]"
    targetelement="listen_team" targetmethod="start" />
</salt:listen>

<!-- call control section -->
<salt:smex id="telephone" sent="start_listening">
  <salt:param name="server" value="ccxmlproc" />
  <salt:bind targetelement="uid" value="/@uid" />
</salt:smex>
2.6.2 Prompt examples

2.6.2.1 Prompt control example
The following example shows how control of the prompt using the methods above might be authored for a platform which does not support a keyword bargein mechanism. On detection of a speech input event, the application reduces the volume of the prompt being played while the input speech is being recognized. The prompt is stopped if recognition succeeds, or is restored to full value if it fails.

```html
<title>Prompt control</title>
<head>
<script>
function checkKWBargein() {
    if (keyword.value == "") { // result is below threshold
        news.change(1.0, 2.0); // restore the volume
        keyword.Start(); // restart the recognition
    } else {
        PromptQueue.Stop(); // keyword detected! Stop the prompt
        // Do whatever that is necessary
    }
}
</script>
<script for="window" event="onload">
news.Start(); keyword.Start();
</script>
</head>
<body>
<salt:prompt id="news" bargein="false" onbargein=" news.change(1.0, 0.5);" >
    <!-- onbargein... turns down the volume while verifying -->
    Stocks turned in another lackluster performance Wednesday as investors received little incentive to make any big moves ahead of next week's Federal Reserve meeting. The tech-heavy Nasdaq Composite Index dropped 42.51 points to close at 2156.26. The Dow Jones Industrial Average fell 17.05 points to 10866.46 after an early-afternoon rally failed.
    <!-- More to follow -->
</salt:prompt>
<salt:listen id="keyword" reject="70"
    onreco="checkKWBargein()"
onnoreco="checkKWBargein()" >
    <salt:grammar src="grams/news_bargein_grammar.grxml" />
</salt:listen>
</body>
</html>
```

2.6.2.2 Using bookmarks and events
The following example shows how bookmark events can be used to determine the semantics of a user response – either a correction to a departure city or the provision of a destination city – in terms of the timing of the bargein during the prompt output. The onbargein handler calls a script which sets a global mark variable to the last bookmark encountered.
Speech Application Language Tags (SALT) 1.0

in the prompt, and the value of this mark is used in the listen’s postprocessing function (ProcessCityConfirm) to set the correct value.

```javascript
var mark;
function interrupt() {
    mark = event.srcElement.bookmark;
}
function ProcessCityConfirm() {
    PromptQueue.stop(); // flush the audio buffer
    if (mark === "mark_origin_city")
        txtBoxOrigin.value = event.srcElement.value;
    else
        txtBoxDest.value = event.srcElement.value;
}
```

2.6.2.3 Prompt playback during page transitions

This example shows how the PromptQueue is used to ensure seamless prompt playback across page transitions in an HTML profile:

```html
<body>

    <form id="form1" action="nextpage.html">
        <input type="button" onclick="transition();" value="go to next page" />
    </form>
    <salt:prompt id="transitionPrompt">
        Let's go to the next page!
    </salt:prompt>
    
    <script>
        function transition() {
            transitionPrompt.Queue();
            PromptQueue.Start();
            form1.submit();
        }
    </script>

</body>
</html>
```

The prompt will play back during the transition to the next page, because although the transitionPrompt element itself is destroyed when the DOM is torn down for the next page, it has been queued onto the PromptQueue object which
is persistent across pages. Subsequent prompts from the next page will be played once the transitionPrompt prompt has been played out.

Notice that the transition() function could be replaced by the simple shorthand transitionPrompt.Start() in the onclick event handler.

2.6.2.4 Queuing prompt subqueues in advance

Certain applications may require not only the content of prompts to be prefetched, but also the earliest possible queuing of prompts in advance of playback, in the interests of efficient operation. The following example shows how individual prompts (or subqueues) which are known to follow the current prompt can be queued while the current prompt is being played back.

This example shows two prompts which are marked with the prefetch attribute (to indicate to the browser that their content should be retrieved at an early opportunity). The first is queued for playback on page load. The second is queued as soon as the first is scheduled for playback by a click on the play button, and is itself played back by a click on the next button.

```
<prompt id="p1" prefetch="true">
    <content href="http://mybank/getStockName.asp?id=1" />
    <content href="http://mybank/getStockValue.asp?id=1" />
</prompt>
<prompt id="p2" prefetch="true">
    <content href="http://mybank/getStockName.asp?id=2" />
    <content href="http://mybank/getStockValue.asp?id=2" />
</prompt>
<body onload="p1.Queue();">
   ...
   <input type="button" value="play" onclick="PromptQueue.Start(); p2.Queue();" />
   <input type="button" value="next" onclick="PromptQueue.Stop(); PromptQueue.Start;"/>
   ...
</body>
```

This model can be scaled up to N prompts by ensuring that the next button always queues the following prompt, by calling the relevant function after the PromptQueue.Start() call, e.g.:

```
<input type="button" value="next"
   onclick="PromptQueue.Stop(); PromptQueue.Start; QueueNextPrompt();" />
```

where QueueNextPrompt() is a function that decides which prompt needs to be queued next.

A simpler solution for the case where the ordering of the prompts is known in advance would be to call Start on each subqueue in sequential order. That is, not only queue each subqueue but also schedule it for playback in advance. This permits the queuing of multiple subqueues in advance, and, in this case, the 'next' functionality would be a simple call to PromptQueue.Stop(), which has the effect of ceasing playback, flushing the current subqueue, and allowing the next to begin playback. (This also allows each subqueue to begin playback immediately after the previous subqueue has finished, without needing to wire the next Start call to the onempty event handler.)

2.6.3 Using SMIL

The following example is taken from section 2.8.3.1.3:

Multimedia prompting followed by recognition:

```
<t:seq>
    <t:part:endsync="all">
        ...
    </t:part>
</t:seq>
```

17 If periods of silence are required either before or during playback, the PromptQueue can be paused while the Start calls are made.
<t:img id="xxx" src="talkinghead.gif"/>
<salt:prompt t:begin="xxx.begin-1s">
Please say the name
</salt:prompt>
</t:par>
<salt:listen> ... </salt:listen>
</t:seq>

The multimedia prompting is provided by an animated GIF and a SALT prompt object. The synthesis is estimated to take one second, therefore the SMIL begin attribute is set to start the synthesis 1 second before the animation. The prompting is contained in a SMIL <par> block, with the endsync attribute set to all. As a result, the following recognition object will not be activated until both the animation and prompt finish playing.

Multimodal activation of recognition or recording:

<input id="clickToTalk" type="button" />
<salt:listen t:begin="clickToTalk.onclick" >
... </salt>

In this example, an HTML button is used to start the listen object.

2.6.4 Wireless Phone (WML) example

The following example demonstrates how SALT speech capabilities might be integrated with WML 1.1 markup for wireless telephones. The example shows the use of voice and/or the keypad to lookup a telephone number (via a server-side script /cgi-bin/lookup.cgi) from a dialing directory by a name (names are assumed to be stored in namelist.grxml).

Notice that this example uses SALT in "declarative mode" because browsers prior to WML 3.0 (the bulk of today’s deployed telephones) do not include WMLScript. The assumption is that SALT elements would be activated by the browser on loading of the relevant card.

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml xmlns:salt="http://www.saltforum.org/2002/SALT">
<head>
<meta http-equiv="Cache-Control" content="max-age=0"/>
</head>
<card id="Splash">
<do type="accept" label="Lookup">
<go href="/cgi-bin/lookup.cgi" method="post">
<postfield name="name" value="$friend"/>
</go>
</do>
<p>
This is the SALT WAP Phone Book. <br/>
<salt:prompt>
Please speak the name of the person
whose number you'd like to find,
</salt:prompt>
or enter from the keypad here:
<input name="friend" title="Name"/>
<salt:listen>
<salt:grammar src="namelist.grxml"/>
<salt:bind targetelement="friend" value="/name[1]"/>
<salt:bind targetelement="Lookup" targetmethod="Click"/>
</salt:listen>
</p>
</card>
</wml>
Such an application would be even more useful if the results of submitting this document would allow a telephone call to be placed using the retrieved number and SALT Call Control. Most of today’s wireless telephones lack the ability to run WML scripts and perform telephony functions concurrently, although this is planned for future generations of equipment.

### 2.6.5 A ‘safe’ voice-only dialog

This example shows prompt and listen elements used with script in a simple voice-only dialog. Its point is to show that all possible user input and error events are caught and safely handled, so that the dialog is never left in a ‘hanging’ state.

```html
<head>
  <title>origin and destination</title>
</head>
<body>
  <form id="travelForm" action="http://mysite.com/travel/inquire.php" method="post">
    <input name="txtBoxOriginCity" type="text" />
    <input name="txtBoxDestCity" type="text" />
  </form>
  <!-- SALT -->
  <salt:prompt id="askOriginCity" onerror="procError()">
    Where from?
  </salt:prompt>
  <salt:prompt id="askDestCity" onerror="procError()">
    Where to?
  </salt:prompt>
  <salt:prompt id="notUnderstood" onerror="procError()">
    Sorry, I could not understand your input.
  </salt:prompt>
  <salt:prompt id="operator" oncomplete="transferToOperator()" onerror="transferToOperator()">
    <!-- external function -->
    I am transferring you to an operator.
  </salt:prompt>
  <salt:listen id="recoOriginCity" onreco="procOriginCity()" onnoreco="procNothingUnderstood()" onsilence="procNothingUnderstood()" onerror="procError()">
    <salt:grammar src="/city.grxml" />
  </salt:listen>
  <salt:listen id="recoDestCity" onreco="procDestCity()" onnoreco="procNothingUnderstood()" onsilence="procNothingUnderstood()" onerror="procError()">
    <salt:grammar src="/city.grxml" />
  </salt:listen>
  <!-- scripts -->
  <script>
    function RunAsk() {
      if (txtboxOriginCity.value=="") {
        askOriginCity.Start();
        recoOriginCity.Start();
      } else if (txtboxDestCity.value=="") {
        askDestCity.Start();
        recoDestCity.Start();
      } else {
        <!-- all slots filled -->
        travelForm.submit();
      }
    }
  </script>
</body>
</html>
```
function procOriginCity () {
    txtBoxOriginCity.value = recoOriginCity.value;
    RunAsk();
}

function procDestCity () {
    txtBoxDestCity.value = recoDestCity.value;
    RunAsk();
}

function procNothingUnderstood() {
    notUnderstood.Start();
    RunAsk();
}

function procError() {
    operator.Start();
}

function terminate() {
    <!-- caller hung up -->
    window.close();
}

<!-- on page load -->
<script>
    <!-- detect disconnect at a central place instead of placing disconnect detect handlers in the listen objects -->
    callControl.attachEvent("call.disconnected", terminate());
    <!-- start dialog execution -->
    RunAsk();
</script>

</body>
</html>

2.6.6 smex examples

2.6.6.1 Logging

<salt:smex id="logServer">
    <salt:param name="d:server" xmlns:d="urn:Microsoft.com/COM">
        <d:protocol>DCOM</d:protocol>
        <d:clsid>209309302930202932098432098</d:clsid>
        <d:iid>0903859304903498530985309094803</d:iid>
    </salt:param>
</salt:smex>

<salt:listen>
    // other directives binding listen results to input fields
    <salt:bind targetelement="logServer" targetattribute="sent" value="*[@log $ge$ 3]"/>
</salt:listen>

This example demonstrates how a logging mechanism can be written using a COM object with its class id and interface id. The speech developers attach an attribute log indicating the level of interests for logging to the relevant SML nodes. In the example above, the developer chooses to log all nodes with log value greater or equal to 3 by using a single bind directive. The example works in both downlevel and uplevel browsers.

The example also intends to demonstrate it is possible for a page to contain multiple smex objects which communicate with the same platform component as long as there will not be confusion on which smex object is responsible for delivering the platform messages back to the SALT document. The above example implies a component can implement multiple interfaces, each of which has its own smex conduit. The same argument could apply to TCP servers listening to multiple ports.
2.6.6.2 Call control with ECMA 323

The following example demonstrates the use of ECMA 323 in SALT. The main purpose of the example is simply to ask the caller to say a phone number and transfer the call. (References inline to parts of the ECMA-323 specification refer to sections in the documents to be found at [http://www.ecma.ch/ecma1/STAND/ecma-323.htm](http://www.ecma.ch/ecma1/STAND/ecma-323.htm)).

The SALT application can be logically composed of the following sections.

Data for the application:

```html
<input name="transferTarget" />
<input name="callerID" />
<input name="callID" />
<input name="deviceID" />
<input name="monitorObject" type="hidden" value="2234" />
<input name="monitorCrossRefID" />
```

Speech objects in English (only section affected by natural language):

```html
<listen id="recNumber" onreco="procRecNumber()" onnoreco="procNoReco()"
onsilence="procNoReco()">
  <grammar src="..."/>
</listen>

<listen id="recYesNo" onreco="procYesNo()" onnoreco="procNoReco()"
onsilence="procNoReco()">
  <grammar src="..."/>
</listen>

<prompt id="sayWelcome">Hello! Please say the phone number to transfer to. </prompt>
<prompt id="askAgain">Sorry, I missed that. Please say the number again. </prompt>
<prompt id="confirm">Did you say <value href="transferTarget"/>? </prompt>
<prompt id="sayBye">Thank you. Your call is being transferred. </prompt>
<prompt id="tryAgain">The number, <value href="transferTarget"/>, cannot be reached for transfer. Please try again later. </prompt>
```

Speech event handlers (dialog logic) in ECMAScript:

```javascript
function procRecNumber() {
  var msg = event.srcElement.recoresult;
  transferTarget.value = msg.SelectSingleNode("*/phoneNumber").value;
  // read recognized phone number
  var confidence = msg.selectSingleNode("/@confidence").value;
  if (confidence < 0.5) {
    confirm.Start(); recYesNo.Start();
  } else {
    sayBye.Start(); ccTransfer();
  }
}

function procYesNo() {
  var answer = event.srcElement.recoresult.SelectSingleNode("*/yes[@confidence>0.5]");
  // accept only yes with confidence
  if (answer == null) {
    procNoReco();
  } else {
    sayBye.Start(); ccTransfer();
  }
}
```
Speech Application Language Tags (SA LT) 1.0

```javascript
function procNoReco() {  
    transferTarget.value = "";
    askAgain.Start(); recNumber.Start();
}
-->
```

The call control section (unaffected by locale, dialog logic):

```xml
<smex id="callControl" onreceive="ccHandler()">...
<script>!-->

// The ccHandler handles the ECMA-323 events.
// Once the connection is answered, a welcome prompt
// is played and the transfer target telephone number is solicited.
// When the speech event handler detects and confirms the correct
// speech input, an ECMA-323 SingleStepTransfer service is used to
// transfer the caller to the new transfer target.

function ccHandler() {
    var msg = event.srcElement.received;
    if (msg.nodeName == "DeliveredEvent") {  // incoming call notification
        if (msg.nodeName == "DeliveredEvent") { // incoming call notification
            // If the connection is alerting (DeliveredEvent, ECMA-323, 15.2.5) the
            // connection information from the Delivered event is saved
            // called.value and deviceID.value) and the call is answered by using the
            // ECMA-323 AnswerCall service with the saved connection information.
            // If the application needed the ANI and DNIS, it could also obtain
            // this information from this event.

            callID.value = msg.selectSingleNode("./connection/callID").value;
            deviceID.value = msg.selectSingleNode("./connection/deviceID").value;
            ccAnswer();
        } else if (msg.nodeName == "EstablishedEvent") {  // call answered
            // Once the connection is answered (EstablishedEvent, ECMA-323, 15.2.8)
            // a welcome prompt is played and the transfer target telephone number
            // is solicited.

            callerID.value = msg.selectSingleNode("./callingDevice/DeviceIdentifier").value;
            sayWelcome.Start(); recNumber.Start();
        } else if (msg.nodeName == "TransferredEvent") {  // call transferred
            // The TransferredEvent (ECMA-323, 15.2.18) is received when
            // the transfer has been completed. ccCleanup is called to clean up
            // the application data.

            ccCleanUp();
        } else if (msg.nodeName == "ConnectionClearedEvent") {  // user hang up
            // A user hang up is indicated by a ConnectionClearedEvent (ECMA-323,
            // 15.2.4) which flushes the prompt queue and cleans the application
            // data. This could happen at any time during the call.

            promptQueue.Flush();
            ccCleanUp();
        } else if (msg.nodeName == "CSTAErrorCode") {  // service failure event
            // The ccError function handles any failure responses from any of the
            // ECMA-323 services that may have failed.
```

© SA LT Forum founding companies page 71
function ccTransfer() { // transferring a call
    // The SingleStepTransferCall service (ECMA-323, 15.1.24) is used to
    // invoke the transfer. There are two elements provided. The first
    // element is the connection information that was obtained from
    // the DeliveredEvent. The second element is the transfer target that
    // was solicited from the caller.
    //
    callControl.sent = "<SingleStepTransferCall" +
        " xmlns='http://www.ecma.ch/standards/ecma-323/csta'>" +
        "<activeCall><callID>" +
        callID.value + "</callID><deviceID>" +
        deviceID.value + "</deviceID><activeCall><transferredTo>" +
        transferTarget.value + "</transferredTo></SingleStepTransferCall>";
}

function ccStartListening() { // listening for call events
    // The MonitorStart service (ECMA-323, 13.1.2) is used to place a
    // monitor on a device so that events can be generated when activity
    // happens at that device. The single element provided indicates the
    // identifier of the device that is to be monitored. In this example
    // it was part of the application data.
    //
    callControl.sent = "<MonitorStart" +
        " xmlns='http://www.ecma.ch/standards/ecma-323/csta'>" +
        "<monitorObject><deviceObject>" +
        monitorObject.value + "</deviceObject></monitorObject></MonitorStart>";
}

function ccAnswer() { // answering a call
    // The AnswerCall service (ECMA-323, 15.1.3) is used to answer the
    // alerting connection. The single element provided is the connection
    // information that was obtained in the Delivered event.
    //
    callControl.sent = "<AnswerCall" +
        " xmlns='http://www.ecma.ch/standards/ecma-323/csta'>" +
        "<callToBeAnswered><callID>" +
        callID.value + "</callID><deviceID>" +
        deviceID.value + "</deviceID></callToBeAnswered></AnswerCall>";
}

function ccCleanUp() {
    callerID.value = ""; transferTarget.value = ""; callID.value = "";
    recNumber.Stop(); recYesNo.Stop(); ...
}

function ccHangup() { // clearing a connection
    // The ClearConnection service (ECMA-323, 15.1.8) is used to clear
    // a connection. In this example, this is used when the transfer is
    // unable to be completed.
    //
    callControl.sent = "<ClearConnection " +
        " xmlns='http://www.ecma.ch/standards/ecma-323/csta'>" +
        "<connectionToBeCleared><callID>" +
        callID.value + "</callID><deviceID>" +
        deviceID.value + "</deviceID><connectionToBeCleared>" +
        "</connectionToBeCleared></ClearConnection>";
}
function ccError() {

  // The ccError function is called to handle any failure responses to
  // ECMA-323 service requests. If there was an error starting a
  // monitor, the application logs an error. If there was an error
  // response to the SingleStepTransfer service, a message is played
  // for the caller and the connection is cleared.

  var request = callControl.sent.substr(1, 7);
  // read the first 7 characters of service requested

  if (request == "Monitor") { // error starting a monitor
    logMessage("ccError", callControl.sent);
  } else if (request == "SingleS") { // error in transfer
    tryAgain.Start();
    ccHangup();
  } // feel free to handle other errors here

  --> </script>

Putting it all together:

<html>
  ...
  <body>
    // data section here
      // put the speech objects here
    </div>
    // speech event handlers here
    // call control section here
    // finally, when the page is loaded...
    <script>
      ccStartListening();
    </script>
  </body>
</html>

2.6.7 Compatibility with visual browsers

SALT documents can be designed to be compatible with both multimodal browsers and legacy (visual-only) browsers. Because SALT extends and enhances markup languages, rather than altering the behavior of the base markup language, SALT documents can be used by legacy browsers by simply omitting or ignoring the SALT tags.

Dynamically generated web pages can examine the browser's HTTP_USER_AGENT to determine whether to include or omit the SALT tags and any associated scripts. This is discussed in section 2.8.1.12.

It is also possible to create static web pages that work equally well with both legacy browsers and multimodal browsers. Because legacy browsers may not recognize the SALT tags, legacy browsers will ignore them. However, SALT-specific text that is not within a tag (not within angle-brackets), will be displayed by legacy browsers. This includes text that is part of an inline grammar, or part of a prompt, for example. The recommended way to exclude the display of such text in legacy browsers is by encompassing it with the span tag as follows:

<span style="display:none">
  <salt:prompt id="giveBalance" xmlns:ssml="http://www.w3.org/2001/10/synthesis">
    Which city do you want to <emphasis> depart </emphasis> from?
  </salt:prompt>

  <salt:grammar xmlns="urn:microsoft.com/speech/schemas/STGF">
    <grammar>
To prevent SALT scripts in static web pages from interfering with legacy browsers, the scripts should be designed such that they do not fail because the legacy browser does not find an object. Therefore, for static pages, it is recommended that scripts test for the existence of SALT objects before referencing them. For example:

```javascript
function procOriginCity () {
    if (txtBoxOriginCity && recoOriginCity) {
        txtBoxOriginCity.value = recoOriginCity.value;
        RunAsk();
    }
}
```

### 2.6.8 Audio recording example

The following example demonstrates recording audio for a voice mail system.

```html
<!-- HTML -->
<!-- on page load -->
<body xmlns:salt="http://www.saltforum.org/2002/SALT" onload="RunAsk()">
    
    <form id="f1" action="http://www.example.com/savewaveform.aspx" method="get">
        <input name="vmail" type="hidden" />
    </form>
    
    <!-- Prompts -->
    <salt:prompt id="p_record" oncomplete="l_recordvm.Start()">
        Please speak after the tone. You may press any key to end your recording.
    </salt:prompt>
    <salt:prompt id="p_save">
        Do you want to save this voicemail?
    </salt:prompt>
    
    <!-- listens -->
    <!-- Recording session - max 60 seconds recording -->
    <salt:listen id="l_recordvm"
        initialtimeout="3000"
        endsilence="1500"
        babbletimeout="60000"
        onreco="saveAudio()"
        onnoreco="saveAudio()"
        onsilence="RunAsk()"
    >
        <salt:record />
    </salt:listen>
    
    <!-- listen for capturing whether user wants to save voice mail -->
    <salt:listen id="l_save" onreco="processSave()">
        <salt:grammar src="/yesno.grxml" />
    </salt:listen>

    <salt:dtmf id="d_stop_rec" onreco="saveAudio()">
        <grammar src="/alldigits.grxml" />
    </salt:dtmf>

    <!-- HTML + script controlling dialog flow -->
```
function RunAsk() {
    if (voicemail.value=="") {
        p_record.Start();
    }
}

// Ask user if they are satisfied with their recording
function saveAudio () {
    p_save.Start();
    l_save.Start();
}

// If user is satisfied post file name back to web server
// otherwise start again
function processSave () {
    smlResult = event.srcElement.recoresult;
    origNode = smlResult.selectSingleNode('//answer/text()');
    if (origNode.value == "Yes") {
        vmail.value = l_recordvm.recordlocation;
        f1.submit();
    } else {
        RunAsk();
    }
}
</script>
</body>

2.6.9 Using XPath for DOM queries

The following example demonstrates XPath queries used on the DOM of the example page itself.

<html>
<head>
    <title>
        SALT XPath example
    </title>
</head>
<script language="JavaScript">
<!-- hide from browsers
var xmlDoc;
function doOnLoad() {
    // load MS XML parser
    xmlDoc = new ActiveXObject("MSXML2.DOMDocument");
    xmlDoc.async = false;
    // load my own HTML document as an XML DOM tree
    if (!xmlDoc.load(location.href)) {
        alert("Error loading myself from: " + location.href);
        return false;
    }
    // fill in "first" query ("*" == entire document)
    queryFromTextBox();
    return true;
}
function queryFromTextBox() {
    document.entry_form.txtResults.value = doXPathQuery(document.entry_form.queryBox.value);
    return true;
}
function doXPathQuery(szQuery) {
    // find nodes that match the query
    xmlDoc.setProperty("SelectionLanguage", "XPath");
    var arrayMatches = xmlDoc.selectNodes(szQuery);
    // display results
    var szResults = "[" + arrayMatches.length + " matches]"
</script>
for (i = 0; i < arrayMatches.length; i++) {
    szResults += "\n\n[match " + (i + 1) + "]\n" +
    arrayMatches[i].xml;
}
return szResults;
}

// -->
</script>
<body bgcolor="white" onLoad="javascript:doOnLoad();">
<form name="entry_form">
Enter XPath query:
<input type="text" name="queryBox" value="*" />
<input type="button" name="queryButton" value="Query Now"
onClick="javascript:queryFromTextBox();" />
<br />
XPath query results (XML format):
<br />
<textarea name="txtResults" cols="100" rows="20"></textarea>
<br />
</form>
</body>
</html>

The following queries are instructive:

//form
//input
//input[@type="text"]

The way in which an XPath query would be made on the callControl object's capabilities property to determine whether the implementation supported the transfer method would be something like this:

var xmlDoc = new ActiveXObject("MSXML2.DOMDocument");
xmlDoc.async = false;
if (!xmlDoc.loadXML(callControl.capabilities)) {
    // capabilities don't parse as valid XML: abort
} xmlDoc.setProperty("SelectionLanguage", "XPath");
var arrayMatches = xmlDoc.selectNodes("//transfer");
if (arrayMatches.length > 0) // transfer is supported

This would typically be written as a JavaScript function and reused as needed.

2.7 Appendix A: SALT DTD

<!-- DTD for SALT 1.0 (2002) -->
<!ENTITY % boolean "(true | false)"
<!-- should be a float between 0.0 and 1.0 -->
<!ENTITY % confidence.value "CDATA">
<!ENTITY % content.type "CDATA">
<!ENTITY % expression "CDATA">
<!ENTITY % milliseconds "CDATA">
<!ENTITY % object.method "CDATA">
<!ENTITY % listen.mode "(automatic | multiple | single)">
<!ENTITY % script.statement "CDATA">
<!ENTITY % script.variable "CDATA">
<!ENTITY % uri "CDATA">
<!ENTITY % xpath.query "CDATA">
<!ENTITY % xpattern.string "CDATA">

<!ELEMENT bind EMPTY>
<!ATTLIST bind
    targetattribute %script.variable;  "value"
    targetelement %script.variable;  #REQUIRED
2.8 Appendix B: SALT modularization and profiles

2.8.1 Modularization of SALT

This section defines a number of SALT modules for use in different profiles according to device capability and application functionality.

SALT browsers fall into the following main classes of device:

- **Smart Clients**: simple or mobile devices with modest computation power and resources. In this case, the speech capabilities may be achieved using a distributed computing architecture, the devices may have only rudimentary displays, and the browsers may not support scripting. A possible scenario for such devices is an eyes-free/hands-free application where only speech input and output modes are available. Examples include PDA, smart phones, set top boxes, and some automobile navigation systems, etc.

- **Rich Clients**: computing devices of similar capabilities to PCs. Usually, the devices have suitable displays, and UI may be more biased towards a visual design (other than for hands-free, eyes-free applications). Speech-related processing may still be distributed, but a network connection is not mandatory. Examples include desktop, wall, and pocket PCs and some automobile PCs. Rich Clients should have no problem supporting scripting.

- **Telephony Servers**: SALT browsers are running on server-grade computers that process multiple phone calls. The user interface includes speech and/or DTMF. Scripting support is considered a reasonable requirement for this class.

Many functional features in SALT do not make sense in all environments. The purpose of SALT modularization is to classify them into proper categories so that browser implementers have the greatest flexibility and the application developer can enjoy maximum interoperability.
2.8.1.1  Declarative Programming Module
The module contains the `bind` subelement and all its attributes, as defined in 2.2.1.2.

Note that application developers can still enjoy the full SALT functionality on a browser that does not implement this module but supports scripting or SMIL. It is reasonable to allow browsers to claim certain level of compliance without declarative module.

2.8.1.2  Basic Recognition Module
The module contains the `listen` object, all the recognition related properties (e.g., `text`, `recoresult`), the `grammar` and `param` subelements and all their attributes, and all the events and methods, as defined in section 2.2.

Support of this module requires implementing ‘automatic’ mode recognition. Additional support of ‘single’ and ‘multiple’ mode is optional. As noted in 2.2.1, support of this module also requires support for the W3C SRGS, W3C NLSML and, if applicable, W3C SLM (N-Gram) Recommendations.

The module is particularly sensible for smart clients, where basic recognition but not recording is needed.

2.8.1.3  Basic Recording Module
The module contains the `listen` object, all the recording related properties (e.g., `recordlocation`, `recordtype`, `recordduration`, `recordsize`), the `record` and `param` subelements and all their attributes, and all the events and methods defined in section 2.2.8.

This module makes sense for browsers that use only input methods that do not generate uncertainties (e.g. DTMF, keyboards, pointing devices). For this case, the browsers should be able to claim certain level of conformance without implementing any speech recognition features.

2.8.1.4  Concurrent Recording and Recognition Module
When a browser claims to support both the Basic Recognition and the Basic Recording modules independently, this does not guarantee that recording and recognition can be performed simultaneously. Note that for distributed recognition, the browser can perform front-end signal processing locally and only send the acoustic features to the recognition servers. Doing so usually lowers the bandwidth requirements considerably. Therefore, there might be cases where recording and recognition are performed by two different remote servers, and the browser only implements single channel streaming but not two-channel multicasting.

Applications can only enjoy simultaneous recording and recognition on a browser supporting this module. This module contains the union of the basic recognition and recording modules. Supporting this module implies the support of both the basic recognition and recording modules.

2.8.1.5  Basic Media Playback Module
The module contains the `prompt` element and all its properties, methods, events, but the `prompt` element can only contain `content` nodes for referring to pre-recorded media files, and not text nodes in speech synthesis markup languages.

2.8.1.6  Speech Synthesis Module
The module contains the `prompt` element and all its properties, methods, events, and subelements\(^\text{18}\), including text nodes for speech synthesis content.

As noted in 2.1.1.1, support of this module also requires support for the W3C SSML Recommendation.

2.8.1.7  Messaging Module
This module contains the `smex` element and all its properties, methods, events, and the `param` subelement, as defined in 2.4.

\(^{18}\) The `PromptQueue` object is not included here. It is a separate module because a multimedia enabled browser (e.g. a SMIL implementation) often has sophisticated mechanisms in place already to synchronize different media types. For that case, synthesized speech should behave more like other media streams that do not define their own media buffer.
2.8.1.8 Call Control Module
This module enables telephony call control. This may be accomplished through the use of `smex` with ECMA-323 messages (section 2.4.4), or by support of the `CallControl` object (section 3).

2.8.1.9 DTMF Module
This module contains the `dtmf` element and all its properties, methods, and events, as defined in 2.3. When this module is supported in addition to the `listen` element (i.e. the Basic Recognition, Basic Recording and/or Concurrent Recognition and Recording modules), the behavior described in section 2.3.6 is required to be supported.

2.8.1.10 PromptQueue Module
The module supports the `PromptQueue` object and all its properties, events, and methods, as defined in 2.1.5 (with the exception of the `Change()` method in 2.1.5.2.4, which is optional).

2.8.1.11 Logging Module
The module contains the global logging function, as defined in 2.5.

2.8.1.12 Run-time determination of supported modules
Given that clients will support differing collections of SALT modules, it is useful for server-side scripts to have a mechanism to determine client capabilities and dynamically generate the appropriate markup.

Traditionally, web servers have been able to examine an environment variable `HTTP_USER_AGENT` to obtain client capability information. Here are some representative examples of `HTTP_USER_AGENT` strings from a cross-section of clients:

- Microsoft Internet Explorer 6.0 on a Windows 2000 desktop PC manufactured by IBM:
  
  Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; T312461; Q312461; .NET CLR 1.0.3328)

- Netscape Navigator 4.72 on a Windows 95 laptop PC manufactured by Sony:
  
  Mozilla/4.7 [en]C-CCK-MCD {Sony} (Win95; U)

- Planetweb browser on a Sega Dreamcast game console:
  
  Mozilla/3.0 (compatible; Planetweb/1.123 JS SSL US Gold; Dreamcast US)

- Openwave WML browser on a Mitsubishi T-250 wireless telephone:
  
  UP.Browser/3.1.03-T250 UP.Link/4.3.3.4

- Palm Web Clipping Application browser on a Palm Pilot VIIx:
  
  Mozilla/2.0 (compatible; Elaine/3.0)

Following this model, a SALT browser will include an identifying substring of the form `SALT X.Y.Z NNNNN` where:

- **X.Y.Z** is the version number `X.Y` of the SALT specification implemented by the browser and the `.Z` portion is a "build number" of that implementation. Example: `1.0.1023` is the 1023rd build of a SALT browser that implements version 1.0 of the SALT specification.

- **NNNNN** is a "bitmap" of the SALT modules supported. This bitmap is the sum (logical OR) of the "bitmap values" for each of the modules as listed in the table below.

<table>
<thead>
<tr>
<th>SALT Module Name</th>
<th>Bitmap Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scripting (e.g., ECMAScript or WMLScript)</td>
<td>1</td>
</tr>
<tr>
<td>Declarative Programming</td>
<td>2</td>
</tr>
<tr>
<td>Basic Recognition</td>
<td>4</td>
</tr>
<tr>
<td>Basic Recording</td>
<td>8</td>
</tr>
<tr>
<td>Concurrent Recognition &amp; Recording</td>
<td>16</td>
</tr>
<tr>
<td>Basic Media Playback</td>
<td>32</td>
</tr>
<tr>
<td>Speech Synthesis</td>
<td>64</td>
</tr>
<tr>
<td>Messaging</td>
<td>128</td>
</tr>
<tr>
<td>Call Control</td>
<td>256</td>
</tr>
<tr>
<td>DTMF</td>
<td>512</td>
</tr>
<tr>
<td>Prompt Queue</td>
<td>1024</td>
</tr>
</tbody>
</table>

19 Trade names and brands are the property of their respective holders.
If the first example HTTP_USER_AGENT string were re-written to include the SALT substring, it would look something like:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; T312461; Q312461; .NET CLR 1.0.3328; SALT 1.0.1023 3583)
```

This client supports all the SALT Modules except "DTMF" (3583 = 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 1024 + 2048).

### 2.8.2 SALT/HTML profiles

This section defines the support of HTML and associated environmental features in terms of multimodal and voice-only profiles.

#### 2.8.2.1 HTML multimodal

SALT can be used in HTML multimodal profiles which support a display. In these cases the extent to which HTML is supported depends on the capabilities of the device. The extent of SALT module support is also dependent on the device capability, and this will be reflected in individual profiles. For example, if a scripting module is not supported, then full object mode of SALT is unlikely to be incorporated in the profile.

All the HTML examples in this document can also be used in compact HTML (cHTML) browsers, as described at [http://www.w3.org/TR/1998/NOTE-compactHTML-19980209/](http://www.w3.org/TR/1998/NOTE-compactHTML-19980209/), since the subset of HTML used in the examples falls under the cHTML definition.

#### 2.8.2.1.1 accesskey and style

When SALT is used in HTML profiles which support the two general HTML attributes `accesskey` and `style`, SALT adopts these attributes and their capabilities into the `listen` and `prompt` elements. As described below, this enables simple declarative authoring of the typical functionality required in multimodal applications.

**accesskey**

When the hosting environment supports accesskey, the attribute has the following semantics:

- **listen**: for 'automatic' mode, the `onkeypress` event for the accesskey invokes the `Start()` method.
- **listen**: for 'single' mode, the `onkeydown` and `onkeyup` events invoke the `Start()` and `Stop()` methods, respectively. In other words, the accesskey enables "push-hold-and-talk".
- **listen**: for 'multiple' mode and recording, the `onkeypress` event toggles the `Start()` and `Stop()` methods. In other words, the accesskey enables "click to talk".
- **prompt**: the `onkeypress` event invokes the `Start()` method.

This permits simple declarative statements such as:

```
<listen accesskey="*" ... />
<prompt accesskey="*" ... />
```

where the `onkeypress` event from the "*" will have the behavior described above without the need for programmatic script activation.

**style**

When the hosting HTML environment supports the style module, the listen object shall at minimum implement that portion of the object model conforming to W3C CSS level 1 specification. The `onclick`, `onmousedown`, `onmouseup` events assume the same behaviors as the `onkeypress`, `onkeydown` and `onkeyup` events as those defined for the `accesskey` above. In addition, when the hosting environment supports `tabindex`, a `listen` object shall have the same behavior as other visual HTML objects.

### 2.8.2.2 HTML voice-only

#### 2.8.2.2.1 HTML module support
This section describes the subset of HTML elements to be supported by a SALT voice-only browser. The subset is defined on the basis of useful functionality in structuring and executing a web application with a SALT speech interface but without a visual display.\footnote{The subset of elements listed here does not correspond strictly to W3C’s existing XHTML Abstract Modules as defined at http://www.w3.org/TR/xhtml-modularization/, since many modules contain elements and functionality superfluous to speech functionality.}

### 2.8.2.2.1.1 XHTML Modules

The following XHTML modules as defined at [http://www.w3.org/TR/xhtml-modularization/abstract_modules.html](http://www.w3.org/TR/xhtml-modularization/abstract_modules.html) should be supported by voice-only XHTML browsers according to the table below. Required elements are in bold typeface, with hyperlinks to the relevant W3C module definition recommendation. (see the following subsection 2.8.2.2.1.1.1 for finer detail on the required level of support for each element).

<table>
<thead>
<tr>
<th>Required</th>
<th>Module Name</th>
<th>Supported elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part</td>
<td><strong>Attribute Collections</strong></td>
<td>This defines the following common attributes: \texttt{class}, \texttt{id}, \texttt{title}, \texttt{xmlns}, \texttt{xml:lang}, style, and common events collection (i.e. onclick, onkeypress, etc)</td>
</tr>
<tr>
<td>All</td>
<td><strong>Structure Module</strong></td>
<td>\texttt{body}, \texttt{head}, \texttt{html}, \texttt{title}</td>
</tr>
<tr>
<td>Part</td>
<td><strong>Text Module</strong></td>
<td>abbr, acronym, address, blockquote, \texttt{br}, \texttt{cite}, \texttt{code}, \texttt{dfn}, \texttt{div}, \texttt{em}, \texttt{h1}, \texttt{h2}, \texttt{h3}, \texttt{h4}, \texttt{h5}, \texttt{h6}, \texttt{kbd}, \texttt{p}, \texttt{pre}, \texttt{q}, \texttt{samp}, \texttt{span}, \texttt{strong}, \texttt{var}</td>
</tr>
<tr>
<td>All</td>
<td><strong>Hypertext Module</strong></td>
<td>\texttt{a}</td>
</tr>
<tr>
<td>No</td>
<td>List Module</td>
<td>dl, dt, dd, ol, ul, li</td>
</tr>
<tr>
<td>No</td>
<td>Applet</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Text Extension</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Presentation</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Edit</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Bi-directional</td>
<td></td>
</tr>
<tr>
<td>No (see below)</td>
<td>Basic forms</td>
<td></td>
</tr>
<tr>
<td>Part</td>
<td><strong>Forms Module</strong></td>
<td>button, fieldset, \texttt{form}, \texttt{input}, \texttt{label}, \texttt{legend}, \texttt{select}, \texttt{optgroup}, \texttt{option}, \texttt{textarea}</td>
</tr>
<tr>
<td>No</td>
<td>Basic Tables</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Table Module</td>
<td>caption, col, colgroup, table, tbody, td, tfoot, th, thead, tr</td>
</tr>
<tr>
<td>No</td>
<td>Image</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Client-side Image Map</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Server-side Image Map</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Frames</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Target</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Iframe</td>
<td></td>
</tr>
<tr>
<td>Part</td>
<td><strong>Intrinsic Events Module</strong></td>
<td>Events attributes (\texttt{onreset}, \texttt{onsubmit} for \texttt{form}, and \texttt{onload}, \texttt{onunload} for \texttt{body})</td>
</tr>
<tr>
<td>All</td>
<td><strong>Metainformation Module</strong></td>
<td>\texttt{meta}</td>
</tr>
<tr>
<td>Part</td>
<td><strong>Scripting Module</strong></td>
<td>noscript, \texttt{script}</td>
</tr>
<tr>
<td>No</td>
<td>Style Sheet</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Style Attribute</td>
<td></td>
</tr>
<tr>
<td>All</td>
<td><strong>Link Module</strong></td>
<td>\texttt{link}</td>
</tr>
<tr>
<td>All</td>
<td><strong>Base Module</strong></td>
<td>\texttt{base}</td>
</tr>
<tr>
<td>No</td>
<td>Name Identification</td>
<td></td>
</tr>
</tbody>
</table>
2.8.2.2.1.1 Elements

The following elements and events must be supported by HTML voice browsers:

- `<!DOCTYPE>`
- `<html>`
- `<head>`
- `<body>`
- `<title>`
- `<div>`
- `<a>`
- `<form>`
- `<input>`
- `<select>`
- `<option>`
- `<textarea>`
- `<meta>`
- `<script>`
- `<link>`
- `<base>`
- `<base>`

Common Events.

The level of support required for the interface of each of the above elements in the supported modules is outlined below. Interfaces which are required are shown in bold. DOM methods and properties (i.e. not attributes) are italicized.

<table>
<thead>
<tr>
<th>ID</th>
<th>Element</th>
<th>Subcategory</th>
<th>Detail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><code>&lt;!DOCTYPE&gt;</code></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>body</code></td>
<td>Attribute set</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>head</code></td>
<td>Attribute set</td>
<td>I18N</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>html</code></td>
<td>Attribute set</td>
<td>I18N</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Attribute</td>
<td>version</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>title</code></td>
<td>Attribute</td>
<td><code>xmlns</code></td>
<td>For XHTML, this attribute defaults to &quot;<a href="http://www.w3.org/1999/xhtml">http://www.w3.org/1999/xhtml</a>&quot;.</td>
</tr>
<tr>
<td>Text</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><code>abbr</code></td>
<td>Attribute</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><code>acronym</code></td>
<td>Attribute</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><code>address</code></td>
<td>Attribute</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td><code>blockquote</code></td>
<td>Attribute</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td><code>code</code></td>
<td>Attribute</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td><code>cite</code></td>
<td>Attribute</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td><code>code</code></td>
<td>Attribute</td>
<td>Common</td>
<td></td>
</tr>
</tbody>
</table>
### Speech Application Language Tags (SALT) 1.0

<table>
<thead>
<tr>
<th>No.</th>
<th>Tag</th>
<th>Attribute</th>
<th>Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>dfn</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>17</td>
<td>div</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>18</td>
<td>em</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>19</td>
<td>h1</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>20</td>
<td>h2</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>21</td>
<td>h3</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>22</td>
<td>h4</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>23</td>
<td>h5</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>24</td>
<td>h6</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>25</td>
<td>kbd</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>26</td>
<td>p</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>27</td>
<td>pre</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>28</td>
<td>samp</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>29</td>
<td>span</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>30</td>
<td>strong</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>31</td>
<td>var</td>
<td>Attribute</td>
<td>common</td>
</tr>
</tbody>
</table>

#### Hypertext

<table>
<thead>
<tr>
<th>No.</th>
<th>Tag</th>
<th>Attribute</th>
<th>Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>a</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>33</td>
<td></td>
<td>Attribute</td>
<td>accesskey</td>
</tr>
<tr>
<td>34</td>
<td></td>
<td>Attribute</td>
<td>charset</td>
</tr>
<tr>
<td>35</td>
<td></td>
<td>Attribute</td>
<td>href</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>Attribute</td>
<td>hreflang</td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>Attribute</td>
<td>rel</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>Attribute</td>
<td>rev</td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>Attribute</td>
<td>tabindex</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>Attribute</td>
<td>type</td>
</tr>
</tbody>
</table>

**DOM method** `click()`  
This allows the simulation of mouse clicks and simpler navigation. Not strictly part of the HTML DOM spec (but is supported in many visual browsers).

#### Forms

<table>
<thead>
<tr>
<th>No.</th>
<th>Tag</th>
<th>Attribute</th>
<th>Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>form</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>43</td>
<td></td>
<td>Attribute</td>
<td>accept</td>
</tr>
<tr>
<td>44</td>
<td></td>
<td>Attribute</td>
<td>accept-charset</td>
</tr>
<tr>
<td>45</td>
<td></td>
<td>Attribute</td>
<td>action</td>
</tr>
<tr>
<td>46</td>
<td></td>
<td>Attribute</td>
<td>method</td>
</tr>
<tr>
<td>47</td>
<td></td>
<td>Attribute</td>
<td>enctype</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td>DOM Property</td>
<td>elements</td>
</tr>
<tr>
<td>49</td>
<td></td>
<td>DOM Property</td>
<td>length</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>DOM method</td>
<td>reset()</td>
</tr>
<tr>
<td>51</td>
<td></td>
<td>DOM method</td>
<td>submit()</td>
</tr>
<tr>
<td>52</td>
<td>input</td>
<td>Attribute</td>
<td>Common</td>
</tr>
<tr>
<td>53</td>
<td></td>
<td>Attribute</td>
<td>accept</td>
</tr>
<tr>
<td></td>
<td>Attribute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-------------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>54</td>
<td>accesskey</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>alt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>checked</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>maxlength</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>readonly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>tabindex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>defaultValue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>click()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>multiple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>tabindex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>options</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>option</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>label</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>selected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>textarea</td>
<td></td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>accesskey</td>
<td></td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>cols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>readonly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>rows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>tabindex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>button</td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>accesskey</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>Tag</td>
<td>Attribute</td>
<td>Value</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>97</td>
<td>Attribute</td>
<td>disabled</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>Attribute</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>Attribute</td>
<td>tabindex</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Attribute</td>
<td>type</td>
<td>(&quot;button&quot;</td>
</tr>
<tr>
<td>101</td>
<td>Attribute</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>fieldset</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>label</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>Attribute</td>
<td>accesskey</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>Attribute</td>
<td>for</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>legend</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>Attribute</td>
<td>accesskey</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>optgroup</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>Attribute</td>
<td>disabled</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Attribute</td>
<td>label</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>a</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>area&amp;</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>frameset&amp;</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>form&amp;</td>
<td>Attribute</td>
<td>onreset</td>
</tr>
<tr>
<td>115</td>
<td>Attribute</td>
<td>onsubmit</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>body&amp;</td>
<td>Attribute</td>
<td>onload</td>
</tr>
<tr>
<td>117</td>
<td>Attribute</td>
<td>onunload</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>label&amp;</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>input&amp;</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>select&amp;</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>textarea&amp;</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>button&amp;</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>meta</td>
<td>Attribute</td>
<td>I18N</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td>Attribute</td>
<td>content</td>
</tr>
<tr>
<td>125</td>
<td>Attribute</td>
<td>http-equiv</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>Attribute</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>Attribute</td>
<td>scheme</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>noscript</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>script</td>
<td>Attribute</td>
<td>charset</td>
</tr>
<tr>
<td>130</td>
<td></td>
<td>Attribute</td>
<td>defer</td>
</tr>
<tr>
<td>131</td>
<td></td>
<td>Attribute</td>
<td>src</td>
</tr>
<tr>
<td>132</td>
<td></td>
<td>Attribute</td>
<td>type</td>
</tr>
<tr>
<td>133</td>
<td></td>
<td>Attribute</td>
<td>xml:space</td>
</tr>
<tr>
<td>134</td>
<td></td>
<td>Attribute</td>
<td>-</td>
</tr>
<tr>
<td>135</td>
<td>link</td>
<td>Attribute</td>
<td>Common</td>
</tr>
</tbody>
</table>
2.8.2.2.1.2 HTML DOM

SALT platforms supporting HTML are expected to support the DOM specified in the HTML DOM Level 1 Core spec (http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/idl-definitions.html).

The following methods however need not be implemented, since they permit application scripts to change the DOM significantly, possibly invalidating internal browser or script host data structures.

- INode insertBefore, replaceChild, appendChild, removeChild
- INamedNodeMap setNameItem, removeNamedItem
- IElement setAttribute, setAttributeNode, removeAttributeNode, normalize
- IText SplitText

2.8.2.2.1.3 Event model

This section contains examples of the event model implemented by (1) the Microsoft Internet Explorer browser (versions 5+), and by (2) the emerging DOM level 2 specification.

2.8.2.2.1.3.1 IE 5,6 event model

Event listener registration:

In HTML, one may use event name like an attribute:

    <listen id="Listen1" onreco="myhandle()"/>

Script method 1: use generic attachEvent method as

    Listen1.attachEvent("onreco", myhandle);

Script method 2: use the event delegate on the object model
Listen1.onreco = myhandle;

All above 3 mechanisms share the same event handler:

```javascript
function myhandle() {
    var obj = event.srcElement;
    // obj is listen object that dispatches the event.
}
```

**Script method 3:** use HTML script tag that registers event listener and implements event handler in one step:

```html
<script for="Listen1" event="onreco" language="Jscript">
    var obj = event.srcElement;
    // obj is the listen object that sends the event.
</script>
```

By definition, an event handler has no argument, returns nothing, and throws no exception.

### 2.8.2.2.1.3.2 DOM Level 2 model

The DOM Level 2 event model is currently specified at [http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/](http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/). This may be used in profiles outside of HTML.

**Event listener registration:**

Since DOM Level 2 HTML module is not finalized yet, currently the standard way to register an event listener is through scripting. A standard `addListener` method, similar to the `attachEvent` above, is defined in the standard for all nodes and can be used as follows:

```javascript
Listen1.addListener("onreco", myhandle, false);
```

The third argument is a Boolean flag indicating whether user wants to initiate capture. See DOM Level 2 for precise definition for event capturing.

As before, an event handler returns nothing and throws no exception, but now has an argument of the event type:

```javascript
function myhandle(event e) {
    var obj = e.target;
    // obj is the object that sends the event.
}
```

Again, refer to the DOM Level 2 documentation for a definition of event type.

### 2.8.2.2.1.4 HTML window object

The following is the proposed subset of features of the window object which is required for implementation by a SALT voice-only browser.

**Methods:**
- `attachEvent`
- `clearInterval`
- `clearTimeout`
- `close`
- `detachEvent`
- `navigate`
- `setInterval`
- `setTimeout`

**Attributes/Properties:**
- `length`
Speech Application Language Tags (SALT) 1.0

- name
- self

Events:
- onbeforeunload (note: this applies to the page)
- onerror
- onload
- onunload (note: this is inherited from HTML object)

Objects:
- clientInformation/navigator
- document
- event
- location

Relevant profiles may also support the PromptQueue object (see section 2.1.5) and/or the CallControl object (Part 3) within the window object.

2.8.2.2.1.5 Using <meta>

Following the principles established for expressing meta data in HTML (see http://www.w3.org/TR/html4/struct/global.html#meta-data), the meta element can be used in SALT to express meta data about the spoken aspects of the page. (This can be used in conjunction with a profile definitions referenced in the HTML head elements).

The content of such data will be meaningful to SALT platforms in proprietary contexts, so it may be considered a page-level equivalent of the param element (which expresses configuration data particular to an individual element). param is defined on the prompt, listen, dtmf and smex elements. Browsers may then treat such data as applicable to the entire page.

The following are sample uses of the meta element in SALT:

```xml
<meta name="recoServer" content="myRecoServer.url" />
<meta name="recoSpeechDetectionThreshold" content="0.15" />
<meta name="promptServer" content="myPromptServer.url" />
<meta name="audioEncoding" content="a-law" />
```

etc.

2.8.2.3 HTML telephony profile

This profile will be defined in terms of the HTML voice-only profile, and the SALT modules specific to conducting telephony dialogs. Relevant profiles may also support the PromptQueue object (see section 2.1.5) and/or the CallControl object (Part 3) within the window object for these profiles.

2.8.3 SALT and SMIL 2.0

This section defines the normative behavior of SALT elements when hosted in a SMIL 2.0 compliant environment. SMIL 2.0 is defined at http://www.w3.org/TR/smil20. SALT modules that are not explicitly described in this section are either unaffected by SMIL 2.0, or a normative behavior is not defined, (e.g., the non-XML SALT modules).

SALT elements may contain no visual presentation. When this is the case, all the presentation-related SMIL attributes attached to SALT elements are ignored.

2.8.3.1 SMIL Timing and Synchronization Module

Below are the desired behaviors when the host language claims support of SMIL 2.0 Timing modules. The basic timing for an element in SMIL consists of specifications on the onset and the duration of the element. This section defines only the onset and duration for the SALT elements. Advanced timing and synchronization semantics of SALT elements will follow SMIL 2.0 specification based on the onset and duration specification in the following sections.

2.8.3.1.1 The listen object
The onset of a `listen` element is the time when its `Start` method is called. As a result, the SMIL `begin` attribute is used to describe when the `Start` method of the `listen` object will be called.

The duration of a `listen` object is the length of audio consumed. The SMIL `min` and `max` attributes are aliased to the `initialtimeout` and `maxtimeout` of the `listen` object, respectively. Effectively, a SMIL `dur` attribute will invoke the `Stop` method of the `listen` object. Note that the end of the audio stream does not mean the recognition or recording results are ready for processing. As a result, it is most often that the SMIL synchronization will be cued off to related `listen` events.

A "freezing" (in SMIL sense) `listen` object simply means the object no longer consumes audio inputs. The recognition or recording process may continue as described above.

(Note: As the `listen` object provides programmatic means for changing grammars, using SMIL to repeat a `listen` object does not always guarantee the same grammar will be used each time. The same applied to SMIL restart. In other words, resetting the state of a `listen` object means to recompile the grammar if the grammar has been modified.)

As of SALT 1.0, a `listen` object cannot be paused and resumed. Since pausing an active element is most frequent inside the SMIL `excl` block, the `listen` object may treat SMIL pause as an alias for invoking the `Cancel` method and the consequent resume as a fresh `Start`.

The SMIL `beginEvent` corresponds to the `listen` object `onspeechdetected` event. The SMIL `endEvent` is raised when the `bind` subelements are processed, or the `listen` object is about to raise `onreco`, `onnoreco`, or `onerror` event.

2.8.3.1.2 The prompt object

Like the `listen` object, the onset is the time when its `Start` method is called.

The duration of a `prompt` object is the length of the audio played plus the time needed to synthesize its textual contents, if any. The `prompt` object shall follow SMIL Content Control Module in resolving streamed audio. Once the text to speech synthesis is finished and streaming audio is resolved per SMIL definitions, the `prompt` object assumes all the behaviors of a SMIL audio object.

The SMIL Prefetch Content Control Module may be used to direct the SALT `prompt` object to synthesize static text prior to its invocation. When used, SALT `prompt` object follows the timing, freshness, and other semantics in SMIL.

2.8.3.1.3 Examples

1) Multimedia prompting followed by recognition:

   `<t:seq>
   <t:par t:endsync="all">
   <t:img id="xxx" src="talkinghead.gif"/>
   <salt:prompt t:begin="xxx.begin-1s">
   Please say the name
   </salt:prompt>
   </t:par>
   <salt:listen> ...</salt:listen>
   </t:seq>`

   The multimedia prompting is provided by an animated GIF and a SALT `prompt` object. The synthesis is estimated to take one second, therefore the SMIL `begin` attribute is set to start the synthesis 1 second before the animation. The prompting is contained in a SMIL `<par>` block, with the `endsync` attribute set to `all`. As a result, the following recognition object will not be activated until both the animation and prompt finish playing.

2) Multimodal activation of recognition or recording:

   `<input id="clickToTalk" type="button"/>
   <salt:listen t:begin="clickToTalk.onclick">`
In this example, an HTML button is used to start the `listen` object.
3 SALT CallControl object

This part specifies the SALT telephony call control object, which can be used in SALT profiles for the control of telephony functionality. (An alternative is to use the smex element with ECMA-323 messages, as described in 2.4.4.)

3.1 CallControl object definition

3.1.1 Requirements

1. An HTML document containing SALT markup must have the ability to provide access to telephony call control related functions, such as answering a call, transferring a call (bridged or unbridged), managing a call or disconnecting a call.
2. The specification must define a means to associate a telephony media stream with SALT media tags, such as tags for Speech Recognition, Recording, Speech Synthesis, and Audio Playback.
3. The call control related objects defined in the specification must provide a programming abstraction that is independent of the underlying call control signaling protocol.
4. The call control related tags and objects defined in the specification must be extensible. Different applications will have varying degrees of need for access to call control functionality from the simple (e.g., interactive voice dialog with a single caller) to the complex (e.g., full call center capability, or enhanced services by service providers). It should be possible for SALT documents to perform run-time query of extension availability to handle variances in the environment.
5. The call control object model specified here should follow an accepted industry standard, and be easy for programmers to use. This approach leverages a trained telephony developer community. This also provides a vision and guidelines for the upgrade path.
6. The call control object model specified here supports first party call control (third party call control is outside the scope of a Speech Recognition endpoint system). The specified model should support both client and server call control requirements.

3.1.2 Solution Overview

The call control object will be specified as an intrinsic entity of the SALT-enhanced browser. Various call control interface implementations conformant with this specification may be "plugged in" by browser-specific configuration procedures and in that way be made accessible to SALT documents. SALT documents can query for the presence of these "plug-ins" at run-time.

The object shall appear in the DOM of the HTML document. The object will have various properties and methods that can be manipulated via ECMAScript code. Some of these methods will create derivative "child" objects, thereby instantiating an entire call control object hierarchy. The objects should also generate events. Event handlers may be written as ECMAScript functions.

The call control object model specification shall be derived from and modeled after the Java Call Processing API (JCP), which is an open industry specification. The SALT call control specification will not necessarily follow those specifications to the letter, as much of those specifications deal with issues specific to the Java language, whereas the SALT call control specification will be adapted to the ECMAScript programming environment in HTML documents.

For call control use examples see section 3.2.

3.1.2.1 Call Control Object Hierarchy

The SALT call control objects comprise a hierarchy.

---

21 A first-party call control system is one where the activating entity (the system executing the call control function) is also one of the parties of the conversation. This is the case with SALT documents, which participate in a dialog with a human. A third-party call control system is one where the activating entity is not one of the parties of the conversation, but is instead a "moderator" of the conversation. This is the typical case with a telephony switching element such as a softswitch, SIP proxy, etc. This specification addresses the former scenario and not the latter.

22 To be explicit, this specification is an ECMAScript binding derived from the documents for JCP/JCC 1.0 (JSR 021) and selected portions of JTAPI 1.4 (JSR 034), which may be located on the World Wide Web at [http://java.sun.com](http://java.sun.com).
Although each of these objects is present, implementation of many of the features is optional. The capabilities property in 3.1.3.2.1 enumerates which of the features is implemented on the current platform. For example, even on a platform that does not support the conferencing feature, the conference object is still present, but it has at most one child call object.

At the very top of the hierarchy are one or more window objects within the browser. Each window object has a document object representing the loaded HTML page. The document object will have all the traditional subordinate objects as defined elsewhere (q.v., W3C HTML DOM at http://www.w3.org).

Each window contains a single callControl object giving a single point of abstraction for interface to the platform's call control functionality.

The callControl object contains one or more provider objects. Each provider allows access to a single telephony implementation.

Different telecommunication vendors can market providers for different styles of telecommunication platforms, so long as they are conformant with this specification. For example, vendor "A" may market a SIP telephony provider for Voice over IP, while vendor "B" may market an ISDN telephony provider for a specific T1/E1 adaptor card.

The SALT call control interface is platform and protocol independent. It provides a common abstraction above many possible telephony platform implementations.

Which providers are present on any given system is a platform browser configuration issue.

Each provider object provides telecommunications connectivity through one or more addresses. In traditional telephone networks, an address is commonly known as a telephone number.

Providers also allow creation and management of conferences, which are logical groupings of calls.23 Conferences may be created and terminated. Calls may be created and terminated, and also moved into or out of conferences. A call is commonly thought of as a "call leg".

Each call has media streams associated with it, known as channels. These media streams will typically be audio, but could also be video streams (for support of video conferencing) or text (for support of teletext or chat).

3.1.2.2 Browser Configuration of Call Control Providers
The method of instantiation of call control providers inside the browser is a platform specific configuration issue.

All other call control objects are derived from the provider object programmatically using methods of call control objects. For example, you can use the createConference() method of a provider object resulting in a conference object. Likewise, you can use the createCall() method of a conference object resulting in a call object, and so on.

23 Those of you intimately familiar with Java call control will notice that this specification is using the terms conference and call in place of call and connection objects, respectively. This is for alignment with the terminology used by the current draft of the Call Control XML (CCXML) document of the W3C Voice Browser Working Group.
The major exception is that incoming calls can result in conference and call objects being spontaneously created by the platform. Appropriate events will be generated informing the script of the creation of those objects, and the names of the newly created objects.

3.1.2.3 Call Control Event Handling
When the browser starts up and each provider object is "plugged in", the first event each provider throws is the "provider.inService" event.

An ECMAScript event handler to catch this event can be written as shown below.

```javascript
<script language="JavaScript">
    callControl.provider[0].attachEvent("provider.inService", myProviderEventHandler);

    function myProviderEventHandler(event, object) {
        if (event == "provider.inService") {
            // handle In-Service event here.
        }
    }
</script>
```

Any call control object can have a programmer-written ECMAScript event handler attached to it using the attachEvent method as illustrated above. The event handler may be dedicated to handling a single event, or multiple events may be attached to a single handler. The handler can discriminate between different arriving events by examining the event parameter passed to the handler. The handler can also tell what call control object threw the event by examining the object parameter.

If any call control object throws an event and it is not caught by a handler attached to the object throwing the event, then the event will "bubble" up to its parent object. The event will continue to "bubble" up the object hierarchy until it is either caught by an attached event handler, or until it ultimately reaches the callControl object, where it will be processed by a system-default event handler.

At a minimum, all events have a srcElement sub-property that refers to the object that generated an event. You can tell, for example, which call was disconnected when you get a call.disconnected event by examining event.srcElement.

Other properties of events depend on the individual event in question.

3.1.2.4 Lifetime of Objects
Objects such as conferences and calls do not spontaneously disappear. For example, a call object does not destroy itself just because of a call disconnect. The programmer must explicitly destroy the object when finished with it.

Objects are persistent regardless of how many windows are opened or closed. All the objects are accessible to any child window of a single browser. The objects are destroyed when the browser exits, however.

The programmer must typically create objects needed before using them. The only exceptions are as follows:

- A conference object and a call object are spontaneously created on an incoming call.
- The programmer has no control over what addresses a provider offers; addresses cannot be created or destroyed, they are essentially a platform provisioning issue.

3.1.2.5 Associating Media Streams with SALT Tags
Because each window is defined to have a single PromptQueue object (for audio output) and single active listen and/or dtmf object (for audio input), the SALT browser implementation will connect the audio input or output to telephony streams as it deems appropriate.

Each telephony media stream is represented by a channel. Each call object typically has two channels: channel[0] for audio output, and channel[1] for audio input. Each conference object also has a channel[0] whose audio is "split out" to each child call of the conference, and a channel[1] comprised of the mixed input audio of all child calls of a conference.
The programmer has some control over which specific audio input channel and/or audio output channel are in use. See the description of mediaSrc and mediaDest properties in the callControl object section below.

Note: Any SALT document that needs to process more than one input stream or output stream concurrently will require the use of multiple windows. The implications of multi-window browsing are still under consideration.

3.1.2.5.1 Associating Telephone Call Disconnect with SALT Tags

When an active telephone call that has associated media streams disconnects (either by the remote end hanging up, or by the local end executing the disconnect() method), the platform will cause the following actions to occur:

- Any active speech objects in the window will be made inactive by having its Stop() method invoked, in this order:
  - All active <listen> objects
  - All active <dtmf> objects
  - In profiles where the PromptQueue is supported, the PromptQueue object
  - In profiles where the PromptQueue is not supported, playback of <prompts> will be stopped.
- A call.disconnected event will be thrown by the call object upon which the disconnect occurred. This event will invoke any attached ECMAScript handlers on that call object, or on any ancestor object in the DOM if the event bubbles up through the DOM (bubbling is the default behavior unless overridden).

3.1.2.6 Support for a Call Distributor

A Call Distributor is an application program that waits for incoming calls and then dispatches sub-programs to service them. In VoiceXML platforms, the Call Distributor behavior is either provided by (i) the platform vendor and inaccessible to the programmer, or (ii) CCXML scripts may be used to program the same functionality.

SALT provides a method of the callControl object named spawn() to assist the coding of a Call Distributor in ECMAScript. Programmers may use this facility if they wish, but they are not required to do so.

The following steps are suggested to implement a Call Distributor behavior:

- Code a SALT document to act as the parent window that waits for incoming calls, and another SALT document to act as the child window to process the call.
- Upon receipt of an incoming call indication (conference.created and call.created events), the callControl.spawn() method may be invoked with an HTML document URL (the child SALT document) as a parameter, and the object ID of the new conference.
- The parent window will "donate" the conference object (and its children) to the child window. The object will be deleted from the DOM of the parent window, and appear in the DOM of the child window. The parent document can go back to listening for more incoming calls. Note that the "donated" conference is not destroyed; it is merely re-parented from one DOM to another.
- The child document will receive the incoming call indication (conference.created and call.created events), as if the call had come into the child window in the first place. ECMAScript code in the child document can now process the call.
- The child window may terminate itself by calling the window.close() method.

3.1.3 Call Control Object Dictionary

The descriptions of object properties below contain abbreviations "R/O" for Read Only and "R/W" for Read / Write. "Read Only" properties can only be examined, not set. "Read / Write" properties may be examined and/or set.

3.1.3.1 Events

All events have at least the following properties, and may have more, depending upon the particular event in question.

3.1.3.1.1 Properties

- cause – R/O – the reason the event was thrown.
- srcElement – R/O – reference to the object that threw the event.
3.1.3.2 callControl Object

The callControl object is the top-level browser object for call control interface. It is a child of each window object.

3.1.3.2.1 Properties

- capabilities – R/O – an XML documentElement listing the functionality supported by the call control implementation. Scripts can use XML DOM methods, e.g., selectNodes(), to discover what capabilities are supported before trying to use them. An implementation that conforms fully to this specification may minimally supply a string value of <conformance version="X.Y" /> where X.Y is the version number of the SALT specification (e.g., "1.0"). If the implementation deviates by subsetting features (e.g., not supporting conferencing) or by extending features (e.g., by adding call center methods to the call object), then this string must represent a valid XML document with all the referred namespaces fully decorated and their schemas publicly discoverable.

- id – R/O – symbolic globally unique id of this object assigned by the platform (URN format).

- mediaDest – R/W – controls where audio output, i.e. the output from the PromptQueue, is heard. If null, output is sent to the device's speaker (or is lost if there is no speaker). If it contains a reference to the output channel of a conference, the audio is heard by all calls in the conference. If it contains the output channel of a call object, the output is sent to and only heard by the specific call referenced, (such a scenario is sometimes referred to as "whisper", in which only one conference participant hears the message). If mediaDest is null and a conference is created, mediaDest is automatically set to that conference object's channel[0]. If mediaDest refers to a conference that is destroyed, mediaDest is automatically set to null. If a prompt is playing while mediaDest changes, the precise timing of when the actual audio switchover takes place is platform-specific. For example, platforms may implement the switchover immediately, at the end of the current prompt, or at the end of all queued prompts. However, the switchover is guaranteed to take place prior to playing a subsequent prompt once the PromptQueue is empty or is stopped.

- mediaSrc – R/W – controls where audio input is sourced for <listen> objects (speech recognition and/or audio recording). If null, input is received from the device's microphone. If it contains a reference to the input channel of a conference object, input is received from a mixture of all of the calls in the conference. If it contains a reference to the input channel of a call object, input is received from only the specific call referenced. If mediaSrc is null and a conference is created, mediaSrc is automatically set to that conference object. If mediaSrc refers to a conference that is destroyed, mediaSrc is automatically set to null. If a <listen> is in progress while mediaSrc changes, the precise timing of when the actual audio switchover takes place is platform-specific. For example, platforms may implement the switchover immediately, at the end of the current <listen>. However, the switchover is guaranteed to take place prior to beginning a subsequent <listen> operation.

- provider[] – R/O -- array of providers configured into the system and accessible through the browser.

- provider.length – R/O -- number of providers configured into the system

3.1.3.2.2 Methods

- spawn(uri, [conf]) -- create a new window object using the URI parameter as the start document, and begin a new sandboxed thread of execution. The new window will have its own callControl object. If the optional conf parameter is specified, it refers to a conference object that the parent window will donate to the new child window. The child window will receive a conference.created event for the conference and a call.created event for each call object that is a child of the conference object being donated. The donated conference and its child objects will be deleted from the DOM of the donating parent window. This is how a parent window can "hand off" a conference and/or call to a child window for processing. Scripts in the child window can be written with the belief that the events represent one or more incoming calls.

3.1.3.2.3 Events

The callControl object does not throw any events; however, it is usually useful to attach an event handler to this object to catch events that bubble up from lower-level objects in the hierarchy.
3.1.3.3 Provider Object

A **provider** represents an abstracted interface to an implementation of a telephony protocol stack; it is the SALT document's "window" into the telephony platform.

Example **providers** can include SS7-ISUP, ISDN, POTS, SIP, and H.323. Vendors may choose to develop one or more of these as separate **providers**, or a single (multi-protocol) **provider** giving an abstracted view of one or more of these.

The **provider** object(s) visible to a SALT document are *instances* of interfaces to the platform's implementation, and when any SALT document manipulates it's own **provider** instance, it does *not* affect the instances of any other running SALT document (i.e., it is a misconception that invoking the **shutdown()** method will shutdown the provider for the entire system).

The methods, properties, and events of **provider** objects and all derivative call control objects are themselves protocol and implementation independent.

### 3.1.3.3.1 State Machine

The **provider** object state machine has three states:

- **InService** -- This state indicates that the **provider** is currently alive and available for use.
- **OutOfService** -- This state indicates that a **provider** is temporarily not available for use. Many methods in this API are invalid when the **provider** is in this state. **Providers** may come back in service at any time (due to system provisioning by the administrator); however, the application can take no direct action to cause this change.
- **Shutdown** -- This state indicates that a **provider** is permanently no longer available for use by this SALT document. Most methods in the API are invalid when the **provider** is in this state. Applications may use the **shutdown()** method on this interface to cause a **provider** to move into the **Shutdown** state.

### 3.1.3.3.2 Properties

- **address[]** -- R/O -- array of **addresses** hosted by the **provider**.
- **address.length** -- R/O -- number of listenable **addresses** hosted by the **provider**.
- **conference[]** -- R/O -- array of **conferences**.
- **conference.length** -- R/O -- number of child **conferences** currently in existence.
- **id** -- R/O -- symbolic globally unique id of this object assigned by the platform (URN format).
- **parent** -- R/O -- the object id of the **callControl** object the **provider** instance is within.
- **state** -- R/O -- the current state of the **provider** object's finite state machine. String value, see section "State Machine" above.

### 3.1.3.3.3 Methods

- **createConference** -- create a child **conference** object.
- **shutdown** -- completely shut down the **provider** (this **provider** instance of a given **window** cannot be restarted). This function performs object memory cleanup in a typical implementation. Some platforms may not need to implement the **shutdown()** function, in which case it silently ignores such calls made by scripts.
3.1.3.4 Events

- **provider.inService** -- the provider is available for use by the script.
- **provider.outOfService** -- the provider is unavailable.
- **provider.shutdown** -- the provider has been shut down.

3.1.3.4 Address Object

**An address** is a connectable endpoint. In order to receive incoming calls, you must listen on a particular address.

In a traditional Public Switched Telephone Network (PSTN) environment, addresses are known as "telephone numbers". They are represented in RFC 2806 compliant URL format, e.g., "tel:+1-888-555-1212".

In Voice over IP (VoIP) environments, addresses are represented as SIP URLs (q.v., RFC 2543) appearing typically like electronic mail addresses (e.g., "sip:fred@flintstone.com") or as H.323 URLs (q.v., RFC 2000) which may appear as electronic mail addresses, simple IP addresses, or free-form gatekeeper aliases (e.g., "h323:barney@rubble.org", "h323:134.128.1.10", or "h323:arbitrary-alias")

How an application registers one or more addresses with a directory service in order to receive incoming calls is beyond the scope of this specification.

Note that applications never explicitly create new **address** objects. Which addresses are available for use is a provider provisioning/configuration issue.

3.1.3.4.1 State Machine

The **address** object has no associated state machine.

3.1.3.4.2 Properties

- **id** – R/O – symbolic globally unique id of this object assigned by the platform (URN format).
- **parent** – R/O -- id of the provider this address is a member of.
- **state** – R/O -- the current "listen" state of the address object. String value, either "Listening" or "Idle".
- **uri** – R/O -- URI of the address. Must be an RFC 2806 URI, a SIP URI, or an H.323 URI.

3.1.3.4.3 Methods

- **listen(state, [answer])** – begin listening for incoming calls on this address (state is True) or stop listening (state is False). This function allows the programmer to have control over exactly which addresses may receive incoming calls, which is useful on platforms (especially servers) that have multiple addresses. Some implementations may choose to automatically listen by default, in which case an explicit call to listen() is not necessary. The optional parameter answer is a boolean indicating whether incoming calls are automatically accepted (value True, the default) so that explicit invocations of accept() are not required; or whether incoming calls must be explicitly accepted or rejected (value False) in order to leave the Alerting state. Note that if you call listen(True), the address will continue listening until you call listen(False), i.e., the listen state is not automatically reset when an incoming call occurs.

3.1.3.4.4 Events

The **address** object throws no events. Incoming calls will generate **conference.created** and **call.created** events.

3.1.3.5 Conference Object

A **conference** is a logical grouping of **calls** that can share their media streams.

---

24 The "callto:" URI namespace as used in Microsoft NetMeeting was never formally registered with the IETF and is deprecated by RFC 2806.
Every call must be created as a member of some conference, even if it is in a conference all by itself. For example, a voicemail application would typically use a conference object with only one call: the person who called in to leave or retrieve messages.

When more than one call is a child of the same conference object, these calls become "conferenced together".

Conference objects are only good for one "lifetime". When the last call leaves a conference, the conference enters the Invalid state. No new calls can enter an Invalid conference. Properties may be examined, and then the conference can be destroyed, and a new one created if needed.

3.1.3.5.1 State Machine

The conference object state machine has three states:
- **Active** – A conference with some current ongoing activity is in this state. Conferences with one or more associated calls must be in this state.
- **Idle** – This is the initial state for all conferences, immediately after creation. In this state, the conference has zero calls.
- **Invalid** – This is the final state for all conferences. Conference objects which lose all of their call objects (via a transition of the last call object into the Disconnected state) moves into this state. Conferences in this state have zero calls and these conference objects may not be used for any future action.

3.1.3.5.2 Properties

- **call[]** – R/O -- array of the calls in the conference.
- **call.length** – R/O -- number of active calls in the conference.
- **channel[]** – R/W – channels of the conference’s media mixer ... channel[0] is the audio output channel which can be used as a mediaDest for <prompt> tags, allowing beeps or intrusion messages to be played into conferences (e.g., “the conference will end in five minutes”) ... channel[1] is the audio input channel which can be used as a mediaSrc for recording, so that you can record the entire conference.
- **channel.length** – R/O – number of channels of the conference.
- **id** – R/O – symbolically unique id of this object assigned by the platform (URN format).
- **parent** – R/O – id of the provider this conference is a member of.
- **state** – R/O -- the current state of the conference object’s finite state machine. String value, see section "State Machine" above.

3.1.3.5.3 Methods

- **createCall()** – create a call as a member of this conference.
- **destroy()** – destroy the conference object. If a conference is in the Active state at the time it is destroyed, the following sequence will occur:
  - All connected child calls are disconnected, resulting in call.disconnected events being thrown. See section 3.1.2.5.1 for a list and order of actions that occur in response to a disconnect.
  - All child call objects are destroyed.
  - A conference.invalid event is thrown, and then the conference object is destroyed.
  - The document script will now get a chance to respond to the pending events. Note that this implies that the call.disconnected event handler will not be able to query the state of the call object because it
has already been destroyed. If this behavior is not desired, then the connected calls should be individually disconnected before destroying the parent conference.

Also note that the conference object will be automatically destroyed if its parent provider is shutdown.

3.1.3.5.4 Events
- conference.active -- the first call has joined the conference.
- conference.created -- the conference has been created.
- conference.invalid -- the last call has left the conference.

3.1.3.6 Call Object
A call is a connection between an external endpoint address and an endpoint address on the platform.

A call can be created in order to place an outgoing call, or may be created as a result of an incoming call.

The external endpoint address is known as the remote.uri and the platform endpoint address is known as the local.uri. They retain these relationships regardless of whether the call was incoming or outgoing (unlike ANI and DNIS who switch senses depending upon the direction of the call).

For calls that have at least one audio output channel and at least one audio input channel, the primary audio output channel is channel[0] and the primary audio input channel is channel[1]. The direction is with respect to the platform upon which SALT is executing.

3.1.3.6.1 State Machine

The call object state machine has seven states:
- Alerting -- This state implies notification of an incoming call.
- Connected -- This state implies that a call is actively part of a telephone call. In common terms, two people talking to one another are represented by two calls of a single conference in the Connected state. A person interacting with a dialog script only requires a single call.
• **Disconnected** -- This state implies the call is no longer part of an active telephone call. A call in this state is interpreted as once previously belonging to this telephone call.

• **Failed** -- This state indicates that a call has failed for some reason. One reason why a call would be in the Failed state is because the destination party of an outgoing call was busy.

• **Idle** -- This state is the initial state for all new calls. Calls which are in the Idle state are not actively part of a telephone call. Calls typically do not stay in the Idle state for long, quickly transitioning to other states.

• **InProgress** -- This state implies that the call object has been contacted by the origination side or is contacting the destination side. The contact happens as a result of the underlying protocol messages. Under certain circumstances, the call may not progress beyond this state. Extension packages elaborate further on this state in various situations.

• **Unknown** – This state implies that the implementation is unable to determine the current state of the call (perhaps due to limitations or latency in underlying signaling). Typically, methods are invalid on calls that are in this state. Calls may move in and out of the Unknown state at any time.

### 3.1.3.6.2 Properties

- **channel[]** – R/O -- array of the channels of the call; channel[0] is the audio output channel which can be used as a mediaDest for <prompt> tags, allowing beeps or messages to be played into calls ... channel[1] is the audio input channel which can be used as a mediaSrc for recording, so that you can record the entire call.

- **channel.length** – R/O -- number of active channels of the call.

- **id** – R/O – symbolic globally unique id of this object assigned by the platform (URN format).

- **local.pi** – R/O – presentation indicator, set as a result of specifying pi in the connect() method. See acceptable values in the table in a section below.

- **local.si** – R/O -- screening indicator, set as a result of specifying si in the connect() method. See acceptable values in the table in a section below.

- **local.uri** – R/W -- URI of the local address endpoint of the call; for incoming calls, this is equivalent to (and may be mapped from) DNIS. The ability to programmatically change local.uri on outgoing calls is provider implementation dependent. This field is in RFC 2806 format.

- **parent** – R/O -- id of the conference this call is a member of.

- **redirect[]** – R/O -- array of redirections of the call (e.g., occurrences of the call being forwarded)

- **redirect.length** – R/O -- length of the redirect[] array, i.e., the number of entries.

- **redirect[].reason** – R/O -- reasons for each of the redirections.

- **redirect[].uri** – R/O -- URI(s) of the intermediate address(es) that redirected the call (e.g., call forwarded); for incoming calls, this is equivalent to (and may be mapped from) RNE. The order of redirect entries is from least recent to most recent: redirect[0].uri is the first number that call was redirected from, and redirect[redirect.length - 1].uri is the last.

- **remote.pi** – R/O – presentation indicator of the remote phone, set as a result of an incoming call or connection of an outgoing call by the connect() method. See acceptable values in the table in a section below.

- **remote.si** – R/O -- screening indicator of the remote phone, set as a result of an incoming call or connection of an outgoing call by the connect() method. See acceptable values in the table in a section below.

- **remote.uri** – R/O -- URI of the remote address endpoint of the call; for incoming calls, this is equivalent to (and may be mapped from) ANI. This field is in RFC 2806 format.

- **state** – R/O -- the current state of the call object’s finite state machine. String value, see section "State Machine" above.

### 3.1.3.6.3 Methods

- **accept()** – answer an Alerting call (in response to receiving a call.alerting event), moving it to the Connected state. Accepting a call will cause a call.connected event.
connect(uri, [pi, si]) – place an outbound call on a call. This is only valid if the call is in the Idle state. The URI parameter is in RFC 2806 format. The presentation indicator pi and the screening indicator si are optional parameters that may be used to control permissions for how caller ID information will be displayed, if the call control implementation supports such functionality. See acceptable values in the table in a section below.

destroy() – destroy the call object. If the call was connected at the time it is destroyed, it will be disconnected first. The call object will be automatically destroyed if any of its ancestor objects in the DOM are destroyed. See section 3.1.2.5.1 for a list and order of actions that occur in response to a disconnect.

disconnect() -- hang up on a call. See section 3.1.2.5.1 for a list and order of actions that occur in response to a disconnect.

join(conference) -- remove call from existing parent conference object and add call to this conference object.

reject([reason]) – reject an Alerting call (in response to receiving a call.alerting event), moving it to the Disconnected state. Rejecting a call will cause a call.disconnected event. The optional reason parameter is a character string describing the reason the call was rejected, it may be one of the following: busyOverflow, queueTimeOverflow, capacityOverflow, calendarOverflow, unknownOverflow. See section 3.1.2.5.1 for a list and order of actions that occur in response to a disconnect.

transfer(uri, [bridge, pi, si]) -- transfer a call from its current endpoint (the telephony platform) to some other destination specified by the URI parameter. The optional bridge parameter is a request that the platform perform a "trombone" transfer (when True) or a "release trunk" transfer (when False). In a "trombone" transfer, the SALT browser remains a party to the call, a new call object is added to the conference object and a call.connected event occurs when the third-party answers. In a "release trunk" transfer, the SALT browser is disconnected from the call and receives a call.disconnected event. The default value is False. The presentation indicator pi and the screening indicator si are optional parameters that may be used to control permissions for how caller ID information will be displayed, if the call control implementation supports such functionality. See acceptable values in the table in a section below.

3.1.3.6.4 Values for Presentation Indicator and Screening Indicator

Presentation Indicator pi: An indicator whether the URI and name fields are allowed to be presented (if available) to the user. This field is optional; if not supported, value is undefined. If supported, the default value is presentation-allowed.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>presentation-allowed</td>
<td>Display URI and name.</td>
</tr>
<tr>
<td>presentation-restricted</td>
<td>Do not display URI and name: show &quot;private&quot;.</td>
</tr>
<tr>
<td>number-lost-due-to-interworking</td>
<td>Information not available for display: show &quot;unknown&quot; or &quot;out of area&quot;.</td>
</tr>
<tr>
<td>reserved-value</td>
<td>Implementation specific.</td>
</tr>
</tbody>
</table>

Screening Indicator si: An indicator of which party or network element has set and/or verified the URI and name fields. This field is optional; if not supported, value is undefined. If supported, the default value is user-provided-unscreened.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-provided-unscreened</td>
<td>The application set URI and name, and has not screened it.</td>
</tr>
<tr>
<td>user-provided-passed</td>
<td>The application set URI and name, has screened it, and it passed screening.</td>
</tr>
<tr>
<td>user-provided-screening-failed</td>
<td>The application set URI and name, has screened it, and it failed screening.</td>
</tr>
<tr>
<td>network-provided</td>
<td>The network set URI and name.</td>
</tr>
</tbody>
</table>

3.1.3.6.5 Events

Please consult the RFC 2806 document for format details. RFC 2806 contains a very rich syntax, including such things as wait-for-dialtone and calling-card DTMF sending for the "tel:" URI, as well as modem dialing strings for the "modem:" URI.
• call.alerting -- an incoming call is "ringing".
• call.connected -- the call has been answered and connected to both local and remote endpoints; for outgoing calls this event may have a type property indicating the type of device that answered (e.g., voice, fax, modem).
• call.created -- the call object was created, either it is an incoming call, or the script explicitly used the createCall() method of a conference object.
• call.disconnected -- the call has been disconnected, either by the remote end, or by the near end using the disconnect() method; this event will have a cause property indicating the disconnect reason and a properties for the call start time and call end time for billing purposes.
• call.failed -- the call has experienced an unexpected failure, or the method could not be performed, e.g., an outgoing call attempt could not connect; this event will have a cause property indicating the failure reason.
• call.inProgress -- an outbound call is in the process of connecting to the remote end.
• call.unknown -- the call is in an unknown state.

The cause property may be one of the following: normal, unknown, busy, callCancelled, destNotObtainable, incompatibleDestination, lockout, resourceNotAvailable, networkCongestion, or networkNotObtainable.

3.2 SALT CallControl illustrative examples

3.2.1 Cooperative call control libraries

The CallControl object can be implemented natively, or it can be implemented as a scripting library that, for example, uses a messaging layer to access underlying call control engines such as ECMA 323 or CCXML. In both cases, applications should use the CallControl object interface directly, and thereby allow interoperability of the application on a wide variety of platforms. Direct use of smex for call control is not necessary in these cases. Where a platform implements additional proprietary features beyond the CallControl object, this should be done by extending the scripting library.

3.2.1.1 CCXML

This example shows how a library script can use CCXML to answer a telephone call, launch a SALT document, and then handle disconnect or transfer requests from the SALT document. This example does not preclude the CCXML interpreter process from running on the same system as the SALT browser or on a different system in a distributed fashion.

The platform supplies the call control library script and corresponding CCXML script. The SALT application programmer is therefore only responsible for supplying the SALT code. For example:

```html
<head>
<title>call control example using CCXML library</title>
</head>
<script language="JavaScript" src="http://vendor/saltcc.js"></script>
<script language="JavaScript">
var caller;
    callControl.attachEvent("call.connected", procConnected);
    callControl.attachEvent("call.disconnected", window.close);
function procConnected(event) { // incoming call
    caller = event.srcElement;
    query_action.Start();
    get_action.Start();
</script>
```
function handleAction() {
    if (action.value == 'disconnect') {
        caller.disconnect();
        window.close();
    } else if (action.value == 'transfer') {
        query_number.Start();
        get_number.Start();
    }
}

</script>
<body>

<salt:prompt id="query_action">
    Do you want to disconnect or transfer?
</salt:prompt>

Action: <input name="action" type="text" value="disconnect" />

<salt:listen id="get_action" onreco="javascript:handleAction();">  
    <salt:grammar>
        <!--
        grammar enables "disconnect | transfer" as input
        and returns this value in "ACTION" node
        -->
    </salt:grammar>
    <bind targetelement="action" value="/ACTION" />
</salt:listen>

<salt:prompt id="query_number">
    What number do you want to transfer to?
</salt:prompt>

Telephone number: <input name="number" type="text" />

<salt:listen id="get_number" onreco='javascript:caller.transfer("tel:" + number.value);();'>
    <grammar src="/telephone_number.grxml" />
    <bind targetelement="number" value="/NUMBER" />
</salt:listen>

<salt:smex id="cc_socket" onreceive="javascript:cc_Receive();">  
    <salt:param name="target">ccxml_server.mycompany.com:7777</salt:param>
    <salt:param name="protocol">SOAP</salt:param>
</salt:smex>

</body>
</html>

Note that other than the declaration of the smex object and of the saltcc.js include script, the application is coded no differently than if the call control object was implemented natively by the platform.

The platform implementation supplies the call control library and corresponding CCXML scripts. For illustrative purposes, only a subset of the complete library and CCXML scripts is shown here. The saltcc.js library implements the call control object methods by calling smex, for example:

function cccalltransfer(uri) {
    var msg = '<event value="salt_request.transfer">
        ' + '<phone_number value="' + number.value + '" />
    '</event>
    cc_socket.sent = msg;
}

function cccalldisconnect(uri) {
    cc_socket.sent = '<event value="salt_request.disconnect" />
}

function call(parent) {  // constructor
    this.parent = parent;
    call.prototype.transfer = cccalltransfer;
call.prototype.disconnect = cccalldisconnect;
}

var cccaller;
function cc_Receive ( ) {
  var event;
  switch (cc_socket.received) {
    case 'salt_response.connected':
      cccaller = new call();
      event.srcElement = cccaller;
      event.reason = "call.connected";
      callControl.fire(event);
      break;
    case 'salt_response.transferred':
      event.srcElement = cccaller;
      event.reason = "call.transferred";
      callControl.fire(event);
      break;
    case 'salt_response.disconnected':
      event.srcElement = cccaller;
      event.reason = "call.disconnected";
      callControl.fire(event);
      break;
  }
}

The CCXML script handles CCXML events and handles smex messages, for example:

```xml
<?xml version="1.0"?>
<ccxml version="1.0">
  <authenticate server="radius.mycompany.com" userid="johnq" password="secret" />
  <var name="salt_sessionid" />
  <eventhandler>
    <transition event="call.CALL_CONNECTED">
      <dialogstart src="mysession.html" type="text/html" />
    </transition>
    <transition event="dialog.started">
      <assign name="salt_sessionid" expr="_event.sessionid" />
      <send target="salt_sessionid" event="salt_response.connected" />
    </transition>
    <transition event="salt_request.disconnect">
      <disconnect />
      <send target="salt_sessionid" event="salt_response.disconnected" />
    </transition>
    <transition event="salt_request.transfer">
      <transfer dest="_event.phone_number" />
      <send target="salt_sessionid" event="salt_response.transferred" />
    </transition>
  </eventhandler>
</ccxml>
```

### 3.2.2 Call Control use case examples

#### 3.2.2.1 Voicemail incoming call

In this example, a caller reaches the number of a network service provider based voice mail service. The service determines whether the caller is the voice mail subscriber or not, and performs the appropriate action.

```html
<head>
<title>Voicemail incoming call</title>
<script type="text/javascript"><![CDATA[
    // Events conference.created and call.created will automatically
    // occur upon incoming calls.
]]>
```
// With autoAnswer=True, the call will automatically
// accept as well, causing call.connected.
// Attach an event handler to catch the incoming call connected event.

callControl.attachEvent("call.connected", procConnected);
function procMailbox() {
  window.navigate("subscriber.asp?mailbox=" + recoMailbox.value);
}
function procConnected(event) {
  // call object that caused this event
  var caller = event.srcElement;
  if (0 == caller.redirect.length) {
    // call dialed into voicemail system directly
    // (was not forward-no-answer)
    if (hasVoicemail(caller.remote.uri)) {
      // subscriber called-in from own office phone,
      // no need to ask for mailbox
      window.navigate("subscriber.asp?mailbox="
          + caller.remote.uri);
    } else {
      // subscriber called-in from another phone
      askMailbox.start();
      recoMailbox.start();
    }
  } else {
    // someone called subscriber, but got forward-no-answer,
    // so now wants to leave a message
    window.navigate("message.asp?mailbox="
        + caller.redirect[caller.redirect.length-1].uri);
  }
}
]
</head>
</body>

3.2.2.2 Notification call
In this example, a notification service dials an outbound call to a subscriber to notify him of a pending dentist appointment.

<head>
<title>Notification call</title>
<script type="text/javascript"><![CDATA[
  var callee, conf;
  function procOnLoad() {
    conf = callControl.provider[0].createConference();
    callee = conf.createCall();
    callee.attachEvent("call.connected", procConnected);
    callee.connect("tel:+1-415-555-1212");
  }
  function procConnected(event) {
    sayReminder.start();
  }
]]>
</script>
</head>
<body onLoad="javascript:procOnLoad()">

<salt:prompt id="sayReminder" oncomplete="javascript:callee.disconnect()">
  Hello, this call is to remind you of your dentist appointment tomorrow.
  Goodbye.
</salt:prompt>
</body>
</html>

3.2.2.3 Notification call with Caller Line Identity set

This is an elaboration of the dentist appointment example, illustrating how to set the Caller ID that would appear on the subscriber's phone.

```html
<head>
<title>Notification call with arbitrary CLI (calling line identity)</title>
<script type="text/javascript">
  var callee, conf;
  function procOnLoad() {
    conf = callControl.provider[0].createConference();
    callee = conf.createCall();
    callee.attachEvent("call.connected", procConnected);
    callee.local.uri = "tel:+1-408-555-1212";
    callee.connect("tel:+1-415-555-1212");
  }
  function procConnected(event) {
    sayReminder.start();
  }
</script>
</head>
<body onLoad="javascript:procOnLoad()">
  <salt:prompt id="sayReminder" oncomplete="javascript:callee.disconnect()">
    Hello, this call is to remind you of your dentist appointment tomorrow. Goodbye.
  </salt:prompt>
</body>
</html>
```

3.2.2.4 Voice Activated Dialing

In this example, a subscriber calls a voice activated dialing service, speaks the number to dial, and the service places the call using network transfer facilities.

```html
<head>
<title>Voice Activated Dialing - Intelligent Network Transfer</title>
<script type="text/javascript">
  var caller;
  callControl.attachEvent("call.connected", procConnected);
  function procConnected(event) { //incoming call
    caller = event.srcElement;
    askPhoneNumber.start();
    recoPhoneNumber.start();
  }
  function procPhoneNumber() {
    caller.transfer("tel:" + recoPhoneNumber.value);
  }
</script>
</head>
<body>
  <salt:prompt id="askPhoneNumber">
    What phone number would you like to dial?
  </salt:prompt>
  <salt:listen id="recoPhoneNumber" onreco="javascript:procPhoneNumber()">
    <salt:grammar src="./phone.grxml" />
  </salt:listen>
</body>
</html>
```
3.2.2.5 Voice Activated Dialing with Active Listen

This is a slightly different example of voice activated dialing. The subscriber calls the service, speaks the number to dial, and is connected using a "trombone" (or "hairpin") of the two call legs in a single conference.

```html
<head>
<title>Voice Activated Dialing - Trombone Conference with Active Listen</title>
<script type="text/javascript">
var caller, callee;
callControl.attachEvent("call.connected", procConnected);
function procConnected(event) {
  // incoming call
  caller = event.srcElement;
  askPhoneNumber.start();
  recoPhoneNumber.start();
}
function procPhoneNumber() {
  callee = caller.parent.createCall();
  callee.attachEvent("call.connected", calleeConnected);
  askHangup.start();
  callee.connect(recoPhoneNumber.value);
  recoHangup.start();
  dtmfPoundPound.start();
}
function calleeConnected(event) { // outgoing call connected
  // note that incoming & outgoing calls now conferenced.
}
function procHangup(callee) { // request to hangup on callee
  // allow caller to place another call
  callee.disconnect();
  askPhoneNumber.start();
  recoPhoneNumber.start();
}
</script>
</head>
<body>
  <salt:prompt id="askPhoneNumber">
    What phone number would you like to dial?
  </salt:prompt>
  <salt:listen id="recoPhoneNumber" onreco="javascript:procPhoneNumber()">
    <salt:grammar src="/phone.grxml" />
  </salt:listen>
  <salt:prompt id="askHangup">
    I am now placing the call. To hang-up, say 'Please hang up now'.
  </salt:prompt>
  <salt:listen id="recoHangup" onreco="javascript:procHangup(callee)">
    <salt:grammar src="/hangup.grxml" />
  </salt:listen>
  <salt:dtmf id="dtmfPoundPound" onreco="javascript:procHangup(callee)">
    <salt:grammar>
      <!-- grammar enabling "##" as input -->
    </salt:grammar>
  </salt:dtmf>
</body>
</html>
```
3.2.2.6 Find Me

This is more elaborate example of the "trombone" scenario above. An arbitrary caller dials a subscriber, the service attempts to contact three locations the subscriber might be at using parallel dialing. When the subscriber answers one of the three lines, he is connected to the caller using a "trombone" conference, and the other two lines are disconnected.

```html
<head>
<title>Find Me - Simultaneously Dial Several Numbers</title>
<script type="text/javascript">
var caller, callee[3], answerer;
var phoneNumber[3];
phoneNumber[0] = "tel:+1-408-555-1212";
phoneNumber[1] = "tel:+1-415-555-1212";
var timeoutID;
callControl.attachEvent("call.connected", procCallerConnected);
function procCallerConnected(event) { //incoming call
  caller = event.srcElement;
caller.attachEvent("call.disconnected", procCallerDisconnected);
  askPleaseWait.start();
  // abort if no answer within 60 seconds
  timeoutID = setTimeout(procTimeout, 60000);
  for (var i = 0; i < phoneNumber.length; i++) {
    var conf = callControl.createConference();
calle[i] = conf.createCall();
calle[i].attachEvent("call.connected", procCalleeConnected);
calle[i].connect(phoneNumber[i]);
  }
}
function procCalleeConnected(event) { // got a callee to answer
  answerer = event.srcElement;
callControl.mediaDest = event.srcElement.channel[0];
callControl.mediaSrc = event.srcElement.channel[1];
  askTakeCall.start();
  recoTakeCall.start();
}
function procTakeCall() {
callControl.mediaDest = caller.parent.channel[0];
callControl.mediaSrc = caller.parent.channel[1];
if (recoTakeCall.value == "yes") {
  clearTimeout(timeoutID);
  // disconnect all other outgoing calls
  for (var i = 0; i < phoneNumber.length; i++) {
    if (answerer != calle[i]) {
      calle[i].parent.destroy();
    }
  }
}
var conference = answerer.parent;
answerer.join(caller.parent); // join outgoing call to incoming
  // call's conference
conference.destroy(); // destroy the now empty outgoing conference
}
function procTimeout() {
  promptQueue.stop();
callControl.mediaDest = caller.channel[0];
  // disconnect all outgoing calls
  for (var i = 0; i < phoneNumber.length; i++) {
    calle[i].parent.destroy();
  }
sayNotAvailable.start();
}
]]>"</script>
```
<salt:prompt id="askPleaseWait">
    Please hold while I attempt to reach him.
</salt:prompt>

<salt:prompt id="askTakeCall">
    Someone is trying to reach you, do you want to take the call?
</salt:prompt>

<salt:listen id="recoTakeCall" onreco="javascript:procTakeCall()">
    <salt:grammar src="./yesno.grxml" />
</salt:listen>

<salt:prompt id="sayNotAvailable" oncomplete="javascript:caller.disconnect()">
    Sorry, he is not available. Goodbye.
</salt:prompt>
4 SALT conformance

This section specifies the conformance criteria for SALT browsers in rendering a SALT document. The SALT modules referenced in this chapter are defined in section 2.8.1.

In this section, uses of the words 'must', 'should' and 'may' are to be interpreted as "MUST" (REQUIRED), "SHOULD" (RECOMMENDED) and "MAY" (OPTIONAL), respectively, as defined in IETF RFC 2119 (http://www.ietf.org/rfc/rfc2119.txt).

Generally speaking, SALT browsers should conform to any interoperability criteria required by the hosting environment. For example:

- SALT platforms should support openly specified Internet application-level protocols, e.g. HTTP 1.1 (IETF RFC 2616), for retrieval of XML or HTML documents containing SALT markup.
- SALT platforms which support the Basic Media Playback module or the Recording module must support ITU G.711 audio encoding (as noted in sections 2.1.1.3 and 2.2.8.1.1, respectively) and may support other standard audio encodings.
- SALT platforms which support the Basic Media Playback module or the Recording module should support audio/basic (IETF RFC 1341), or audio/wav media types for playback and/or recording of audio content, and openly specified Internet protocols for audio transmittal.
- SALT platforms which support the Basic Media Playback module or the Recording module may support RTSP (IETF RFC 2326) and RTP (IETF RFC 1889), if streaming audio playback or recording.
- SALT platforms which support telephony interfaces should support openly specified telephony signaling and media transport protocols, such as standards published by IETF, ITU, ECMA or other organizations.
- SALT platforms which support remote speech synthesis and recognition services should support openly specified standard protocols for remote speech services such as standards published by IETF, W3C, ETSI or other organizations.

4.1 Portable extensibility

A SALT compliant browser must allow standardized extensibility in XML and make publicly discoverable (1) all the namespaces it natively recognizes, and (2) the policy of processing a non-natively recognized namespace, which may range from as simple as ignoring the namespace to as sophisticated as publishing the UDDI providers the browser will use to obtain a list of Web services that can potentially resolve the namespace. A SALT application must be able to ascertain whether a compliant SALT browser can render XML extensions in a SALT document, or the extensions must be translated before a SALT document is served to the browser.

A SALT compliant browser must recognize the namespaces for W3C speech recognition grammar (SRGS) and speech synthesis markup language (SSML) for inline grammar and synthesis markups once they reach W3C Recommendation status. In addition, it is recommended that SALT browsers recognize the specifications for W3C Semantic Interpretation for Speech Recognition specification once it reaches Recommendation.

4.2 Browser types

Within the SALT namespace, the compliance criteria are based on the modularization described above, and can be summarized in the following table where M stands for Mandatory, O for Optional, and N/A for not applicable. A SALT browser must specify which category (column) it claims compliance. A browser claiming compliance to a particular category must support all the mandatory modules for the category, and within each module, all the behavior required in the module definition (section 2.8.1) must be implemented.

For modules which are optional, browsers which implement functionality similar to the functionality provided in the optional module are strongly encouraged to support such functionality exactly according to the module definition, that is to support the SALT module rather than proprietary methods, in order to allow greater portability of applications.

---

27 "Mandatory" and "Optional" should be interpreted as equivalent to "REQUIRED" and "OPTIONAL", respectively, as defined in IETF RFC 2119 (http://www.ietf.org/rfc/rfc2119.txt)
Support for the Call Control module is described in further detail below.

As noted in the definition of the Basic Recognition module (2.8.1.2), "automatic" mode recognition is the minimum level of compliance. Browsers must also make publicly discoverable all recognition modes which are natively implemented.

4.3 Call control support

As indicated in the table above, support of the telephony call control module in SALT is optional for all browser types. However, in order to permit applications which use call control to be portable across browsers, SALT browsers may also make a Portability Claim, as described below.

Portability Claim

In addition to claiming conformance to one of the SALT browser types in 4.2, SALT browsers may claim application portability if the telephony call control functions are provided through either (1) the message formats defined in ECMA-323 using the smex object, or (2) the CallControl object model defined in Part II of the SALT specification.

When a portability claim is made based on ECMA-323, browsers implement the functionality of one or more profiles of ECMA-269 (as defined in section 2.1.3 of the ECMA-269 specification, http://www.ecma.ch/ecma1/STAND/ecma-269.htm), and adhere to the conformance criteria of ECMA 323 (http://www.ecma.ch/ecma1/STAND/ecma-323.htm). Browsers should make publicly discoverable the ECMA-269 profiles which are implemented and the XML namespaces which are recognized.

When portability claim is made based on the CallControl object model, browsers must implement the capability discovery in the CallControl object and make discoverable the XML schema of the capability description. The CallControl object model does not necessarily have to be provided as a browser native feature. As a result, telephony platforms using private messages other than ECMA 323 can claim application portability if a CallControl object library exists to translate the private messages.