SALT

Speech Application Language Tags
0.9 Specification
(Draft)

Document **SALT.0.9.pdf**

19 Feb 2002

This specification is a working draft of the Technical Working Group of the SALT Forum, and is subject to change without notice. The SALT Forum reserves the right to make any changes to this specification prior to final release. It is inappropriate to use draft documents as reference material or to cite them other than as "work in progress."

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

SALT (Speech Application Language Tags) is an extension of HTML and other markup languages (cHTML, XHTML, WML) which adds a spoken dialog interface to web applications, for both voice only browsers (e.g. over the 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 flavours, 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 of recognition results

and <listen> also contains the facility to record audio input

<record …> for recording audio input

A call control object is also provided for control of telephony functionality.

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.

SALT also provides DTMF and call control capabilities for telephony browsers running voice-only applications through a set of DOM objects properties, methods and events.

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 pen-down on a textbox, for example, which activates a grammar relevant to the textbox, and binds the recognition result into the textbox:

```
<xhtml xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
      <!—- HTML -->
      ...
      <input name="txtBoxCity" type="text" onpendown="listenCity.Start()"/>
      ...

      <!—- SALT -->
      <salt:listen id="listenCity">
            <salt:grammar name="gramCity" src="./city.xml" />
            <salt:bind  targetelement="txtBoxCity"
                        value="//city" />
      </salt:listen>
</xhtml>
```

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, eg generic forms of the RunAsk script below or the RunSpeech script illustrated in section 8.1.2).

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:

```
<xhtml xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
      <!—- HTML -->
      ...
      <input name="txtBoxOriginCity" type="text" />
      <input name="txtBoxDestCity" type="text" />
      ...

      <!—- SALT -->
      <salt:prompt id="askOriginCity"> Where from? </prompt>
      <salt:prompt id="askDestCity"> Where to? </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>

      <!—- HTML + script -->
      <script>
            <![CDATA[
            function RunAsk() {
                  if (txtboxOriginCity.value=="") {
                        askOriginCity.Start();
                        recoOriginCity.Start();
                  } else if (txtboxDestCity.value=="") {
                        askDestCity.Start();
                        recoDestCity.Start();
                  }
```

```
                }
                function procOriginCity () {
                      txtBoxOriginCity.value = recoOriginCity.value;
                      RunAsk();
                }
                function procDestCity () {
                      txtBoxDestCity.value = recoDestCity.value;
                }
                ]]>
          </script>

          <!—- on page load -->
          <script><![CDATA[
                RunAsk();
          ]]></script>
    </xhtml>
```

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, eg 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 will be encourged 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 Modes of execution

Since SALT uses to implement its execution model the browser environment of the page in which it is hosted, 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, where the full interface of each SALT element is exposed by 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 candidate recommendation.)

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.

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:

```
<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 should 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 dialog turn selection scripts on the basis of such events.

Each SALT element also 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.

The event handler examples in the document assume that object mode browsers support the HTML event mechanism as implemented in Microsoft's Internet Explorer (v5+) browser. The SALT Forum is also monitoring W3C progress on defining DOM Level 2 events. For an illustration of the IE event model and comparison with the emerging W3C DOM Level 2 event model, see section 10.2.2.1.3.

1.4 Terms and definitions

Term	Definition
Downlevel browser	A browser which does not support full eventing and scripting capabilities. This kind of SALT browser will support the declarative aspects of a given SpeechTag (i.e. rednering of the core element and attributes), but will not expose the DOM object properties, methods and events to direct manipulation by the application. Downlevel browsers will typically be found on clients with limited processing capabilities.
Event bubbling / Event propagation	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.
Mixed Initiative	A form of dialog interaction model, whereby the user is permitted to share the dialog initiative with the system, eg by providing more answers than requested by a prompt, or by switching task when not prompted to do so. (see also System Initiative.)
NLSML	Natural Language Semantic Markup Language 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 http://www.w3.org/TR/nl-spec/. See also SAPI SML.
SAPI SML	SAPI Semantic markup language. The XML document returned by SAPI 6.0 when an utterance is determined to be in-grammar. (SAPI SML is a SAPI-specific return format. SALT interpreters are agnostic to the actual content format of the returned document, provided it is an XML document). SAPI SML contains semantic values, confidence scores and the words used by the speaker. (It is generated by script or XSLT instructions contained within the grammar rules.) See also NLSML.
SRGS	Speech Recognition Grammar Specification. W3C specification for representating speech recognition grammars. The latest version may be found at http://www.w3.org/TR/speech-grammar/
SMIL	Synchronized Multimedia Integration Language. A W3C proposed recommendation, SMIL (pronounced "smile") enables simple authoring of interactive audiovisual presentations. See

Term	Definition
	http://www.w3.org/AudioVideo/.
SSML	Speech Synthesis Markup Language. W3C XML specification for controlling the output of a prompt engine. The latest version may be found at http://www.w3.org/TR/speech-synthesis.
System Initiative	A form of dialog interaction model, whereby the system holds the initiative, and drives the dialog with typically simple questions to which only a single answer is possible. (see also Mixed Initiative.)
Uplevel browser	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.
XPath	XML Path language, a W3C recommendation for addressing parts of an XML document. See http://www.w3.org/TR/xpath.

1.5 Document structure

The next sections, chapters 2 - 5 describe the core elements of the SALT markup: <listen>, <prompt>, <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 6 describes the CallControl object that may be available in telephony profiles for the control of telephony functionality. Chapter 7 describes the logging function for the recording of platform events and data. Chapter 8 contains a number of examples illustrating the use of SALT to accomplish a variety of tasks. Chapter 9 holds the SALT DTD. Chapter 10 introduces modularization and profiling issues and outlines SALT/HTML profiles for multimodal and voice-only scenarios.

2 Speech input: <listen>

The listen element is used to specify possible user inputs and a means for dealing with the input results. As such, its main elements are <grammar> and <bind>, and it contains resources for configuring recognizer properties. listen can also be used for recording speech input, and its <record> subelement is used to configure this process. The activation of listen elements is controlled using Start, Stop and Cancel methods. <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.

The use of the listen element for speech recognition is defined in the following sections. The use of listen elements for recording is described in detail in section 2.8.

2.1 listen content

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 supports the programmatic activation and deactivation of individual grammar rules. Note also that all top-level rules in a grammar are active by default for a recognition context.

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 top-level rules of a listen's grammars are treated as active unless explicitly deactivated.

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/. A SALT browser may support any other grammar formats. Note: the W3C SRGS specification is currently a Working Draft and not yet a W3C Recommendation.

Attributes:

- **name** Optional. This value identifies the grammar for the purposes of activation and deactivation (see 2.3.4 and 2.3.5). Grammars within the same <listen> element should 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. 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 mime-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 type="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/ for usage.
- **xml:lang** Optional. String indicating which language the grammar refers to. The value of this attribute follows the xml:lang definition in RFC3066 of the IETF[1]. 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> (eg <listen>)[2]. 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.

Example referenced and inline grammars:

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

      or

<salt:grammar xmlns="http://www.w3.org/2001/06/grammar">
      <grammar>
          <rule>
              from
              <ruleref name="cities" />
          </rule>
          <rule name="cities">
              <oneof>
                   <item> Cambridge </grxml:item>
                   <item> Seattle   </grxml:item>
                   <item> London    </grxml:item>
              </oneof>
          </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

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 dication 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 listen mode attribute, see section 2.6).

[1] For an explanation of RFC3066 in terms of HTML and XML, see http://www.w3.org/International/O-HTML-tags.html.

[2] 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.

To enable interoperability of SALT applications, SALT browsers that 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: 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.

2.1.2 <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 or SAPI SML) 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: the W3C NLSML specification is currently a Working Draft and not yet a W3C Recommendation.

Sample W3C NLSML and SAPI SML returns for the utterance "I'd like to travel from Seattle to Boston" are illustrated respectively below:

```
<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>

<SML confidence="0.4">
      <travel text="I'd like to travel from Seattle to Boston">
            <origin_city confidence="0.45"> Seattle </origin_city>
            <dest_city   confidence="0.35"> Boston  </dest_city>
      </travel>
</SML>
```

Since an in-grammar recognition is assumed to produce 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 on a successful recognition before the onreco event is thrown, and 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. An ***XML Pattern*** (as in the W3C XML DOM specification) string 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 a succesful recognition.
- **value:** Required. An ***XPATH*** (as in the W3C XML DOM specification) string that specifies the value from the recognition result document to be assigned to the target element.

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

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

```
<xhtml xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
    ...
    <form id="formTravel">
        <input name="txtBoxOrigin" type="text"/>
        <input name="txtBoxDest" type="text" />
    </form>
    ...
    <salt:listen id="listenTravel">
        <salt:grammar src="./city.xml" />

        <salt:bind      targetelement="txtBoxOrigin"
                        value="//origin_city" />
        <salt:bind      targetelement="txtBoxDest"
                        value="//dest_city" />
    </salt:listen>
    ...
</xhtml>
```

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:

```
<salt:bind  targetelement="txtBoxDest"
            value="//dest_city"
            test="//dest_city[@confidence &gt; 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:

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

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 event handler to permit programmatic script analysis and post-processing of the recognition return (see 2.4.1) or recording results (see 2.8.4.1).

Further illustrations of the use of <bind> may be found in the sample markup examples in 8.8.

2.1.3 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 page-level parameter settings may also be defined using the <meta ...> element (see 10.2.2.1.5).

2.2 Attributes and properties

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

2.2.1 Attributes

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

- **initialtimeout**: Optional. The time in milliseconds between start of recognition 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 2.4.2). 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 2.6), 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 speech detection and each 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 (eg by holding down the stylus in tap-and-talk). For a summary of onnoreco status codes, see section 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 and results returned to the browser. If exceeded, an onerror event is thrown by the browser – 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. 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 2.6), 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 recognition rejection threshold 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 RFC3066 of the IETF. 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 2.1.1).
- **mode:** Optional. String specifying the recognition mode to be followed (see 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 2.8.2.1.

In certain HTML profiles, the HTML attributes **accesskey** and **style** may also be used as attributes of listen, as described in 10.2.1.1.

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 2.1.2, In case of no recognition, the return may be empty.
- **text** Read-only. A string holding the text of the words recognized. (In SAPI SML, this string is also contained in the text attribute of the highest level element).
- **status:** Read-only. Status code returned by the recognition platform. The status property is only meaningful after status-setting events are thrown by the listen object, and as such should be examined in the handler of the relevant event. Possible values are 0 for successful recognition, or the failure values -1 to -3 (as defined in the exceptions possible on the Start method (section 2.3.1) and Activate method (section 2.3.4)) and statuses -11 to -15 set on the reception of recognizer error events (see 2.4.6), and statuses -20 to -24 in the case of recording (see 2.8.5.1).

2.3 Object 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.

2.3.1 Start

The Start method starts the recognition process, using as active grammars all the top-level rules for the recognition context which have not been explicitly deactivated. As a result of the start method, a speech recognition event such as onreco, onnoreco, or onsilence will typically be fired, or an onerror or onaudiointerrupt event will be thrown in the cases of a platform error or loss of connection, respectively. See section 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 4.6.)

Syntax:

Object.Start()

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 2.4.5 lists possible non-zero status codes.

On the calling of the Start() method the speech recognition platform should ensure that the active grammars of a <listen> are complete and up-to-date. onerror events resulting from this process are thrown according to the status codes in section 2.4.5.

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. (Note that this method need not be used explicitly for typical recognitions in automatic mode (see 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.

Syntax:
Object.Stop()
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 2.4.4), and if there is any problem an onerror event is fired with and the status codes as outlined in section 2.3.1 are set.

2.3.3 Cancel

The Cancel method stops the audio feed to the recognizer, deactivates the grammar and releases the recognizer. The platform may return a recognition result for a cancelled recognition (although this should 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()
Return value:
None.
Exception:
None.

2.3.4 Activate

The Activate method activates the grammars of a <listen>. The first argument identifies the grammar for activation, the second an optional, 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: all the top-level rules of a listen's grammars are treated as active unless explicitly deactivated.)

Syntax:
Object.Activate(grammarName, [ruleName]);
Parameters:
- **grammarName:** Required. Grammar name.
- **ruleName:** Optional. Rule name.

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.)

2.3.5 Deactivate

The Deactivate method deactivates the grammars of a <listen>. The first argument identifies the grammar for deactivation, the second an optional, 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:** Required. Grammar name.

- **ruleName:** Optional. Rule name.

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.)

2.4 listen events

The listen DOM 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 2.6.

Although browsers will update the recoresult property for both successful and unsuccessful events from the speech recognizer, this property should only be assumed to hold a valid result 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).

2.4.1 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. For listens in automatic mode, this event stops the recognition process automatically and clears resources. The onreco handler is typically used for programmatic analysis of the recognition result and processing of the result into the page.

Syntax:

Inline HTML	<listen onreco **="***handler***" >**
Event property	Object.onreco = *handler*; Object.onreco = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	User says something
Default action	Return recognition result object

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.

```
<xhtml xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
      <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.xml" />
      </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>
        </xhtml>
```

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 2.2.1). This event cancels the recognition process automatically for the automatic recognition mode – see **Figure 1**.

Syntax:

Inline HTML	<listen onsilence="*handler*" …>
Event property (in ECMAScript)	Object.onsilence = *handler* Object.onsilence = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Recognizer did not detect speech within the period specified in the initialtimeout attribute.
Default action	Set status = -11

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

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 2.1.3), 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 3.4.2), and may disable the initial timeout of a started dtmf object, as described in 4.6. 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.

Syntax:

Inline HTML	<listen onspeechdetected="*handler*" …>
Event property (in ECMAScript)	Object.onspeechdetected = *handler* Object.onspeechdetected = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Recognizer detects speech.
Default action	Trigger onbargein if prompt is in playback, disable dtmf initial timeout if started.

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

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.

Syntax:

Inline HTML	<listen onnoreco **="***handler***" >**
Event property	Object.onnoreco = *handler*; Object.onnoreco = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Recognizer detects speech but is unable to fully interpret the utterance.
Default action	Update recoresult and status properties. recoresult may be an empty document or it may hold information provided by the speech recognizer. Status codes are set as follows: **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 2.2.1); **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 2.2.1).

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

2.4.5 onaudiointerrupt:

The onaudiointerrupt event is fired when the communication channel with the user is lost. In telephony scenarios, for example, it will fire when the line is disconnected. Its default action is to stop recognition and return results up to that point (i.e. just as in listen.stop). Its handler is typically used for local clean-up of resources relevant to an individual listen. In relevant profiles, this event is also fired on <dtmf> and the PromptQueue and call control objects. The order of firing is as follows:

1. listen
2. dtmf
3. PromptQueue object
4. call control object.

Syntax:

Event property	Object.onaudiointerrupt= *handler*; Object.onaudiointerrupt = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The system detects the loss of the connection of the communication channel with the user.
Default action	stop recognition and return results.

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

2.4.6 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 info table below.

Syntax:

Inline HTML	<listen onerror **="***handler***" >**
Event property	Object.onerror = *handler*; Object.onerror = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The grammar activation or recognition process experiences a serious or fatal problem.
Default action	Set status property and return null recognition result. The listen's recoresult and text properties are set to empty. Status codes are set as follows: **status -1:** A generic (speech) platform error occurred during recognition. **status -2:** Failure to find a speech platform (for distributed architectures) **status -3:** An illegal property/attribute setting that causes a problem with the recognition request. **status -4:** Failure to find resource – in the case of recognition this is a grammar resource. **status -5:** Failure to load or compile a grammar resource **status -6:** Failure to activate or deactivate rules/grammars (this is thrown as as result of the Activate/Deactivate methods as in 2.3.4, 2.3.5). **status -7:** The period specified in the maxtime attribute (see 2.2.1) expired before recognition was completed

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

2.5 Interaction with DTMF

In telephony profiles which support DTMF input, platforms will implement certain links between speech and dtmf which simplify the authoring of joint behavior on a single dialog turn. This relationship is discussed in section 4.6 of the DTMF input chapter.

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 of the process and the return of results may differ.

For example, an end-user using tap-and-talk in a multimodal application will control the period he wishes talk to the device by tapping and holding a form field, so the browser uses a GUI event, eg pen up, to control when recognition should 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 (here called 'open microphone') 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 will define profiles for the support expected of different modes according to the class of client device. 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
Many 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 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.

2.6.1 Automatic mode

<listen **mode="automatic"** … >

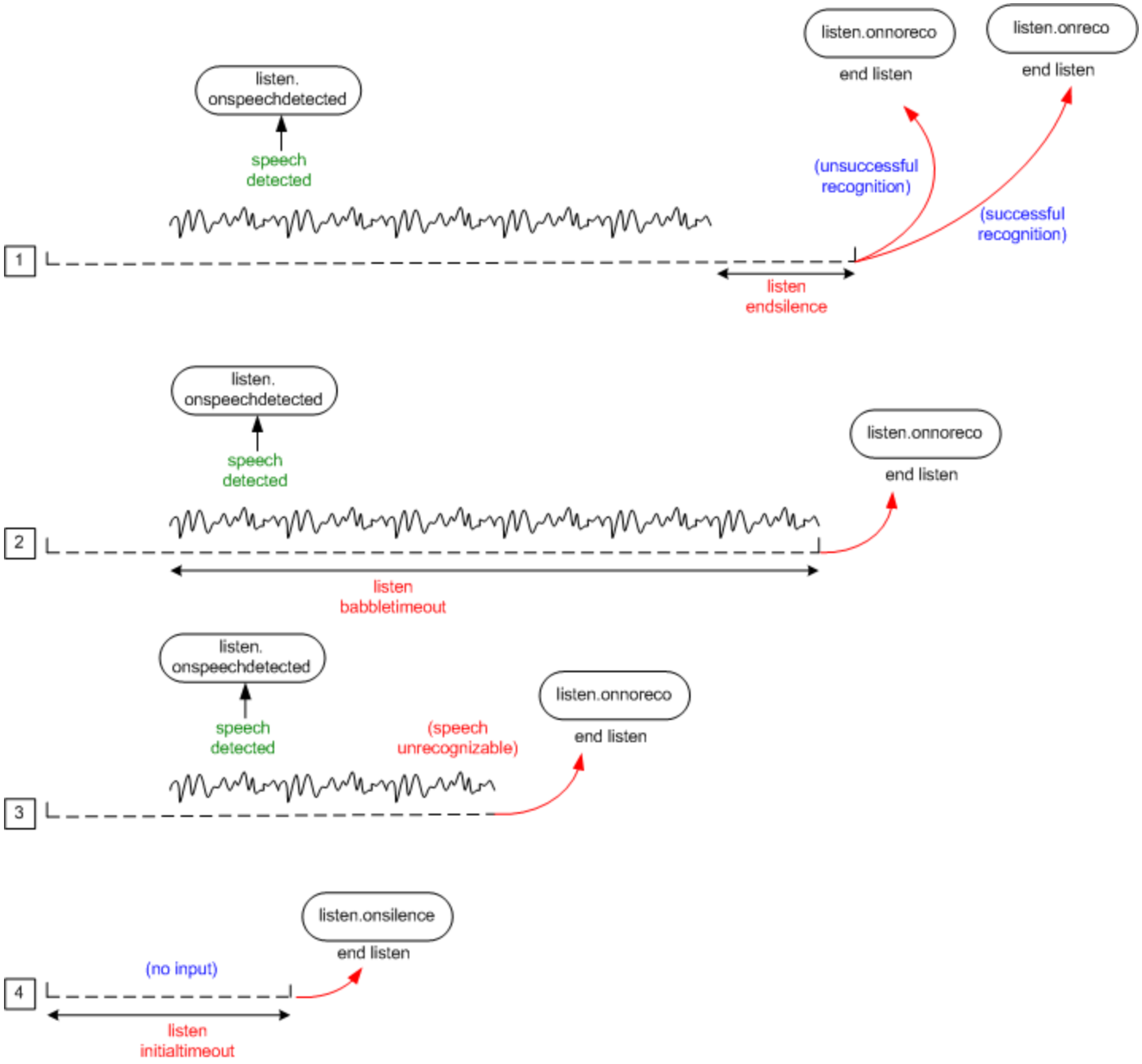


Figure 1: Automatic mode listen timeline

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.

As soon as a recognition is available (the *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 onreco event is thrown for a successful recognition (i.e. confidence higher than the threshold specified in the reject attribute), and onnoreco for an unsuccessful recognition (i.e. confidence higher than the threshold specified in the reject attribute). This is shown in diagrammatic form in case (1) of **Figure 1: Automatic mode listen timeline**. Case (2) shows the firing of onnoreco after the babbletimeout period is exceeded, which ends execution of the listen. Case (3) displays an unsuccesful recognition attempt where the recognizer throws onnoreco before the utterance endsilence. Case (4) shows no input from the user, and

the resulting throwing of the onsilence event. As noted above, all events except onspeechdetected end the execution of a listen in automatic mode.

2.6.2 Single mode

<listen **mode="single"** … >

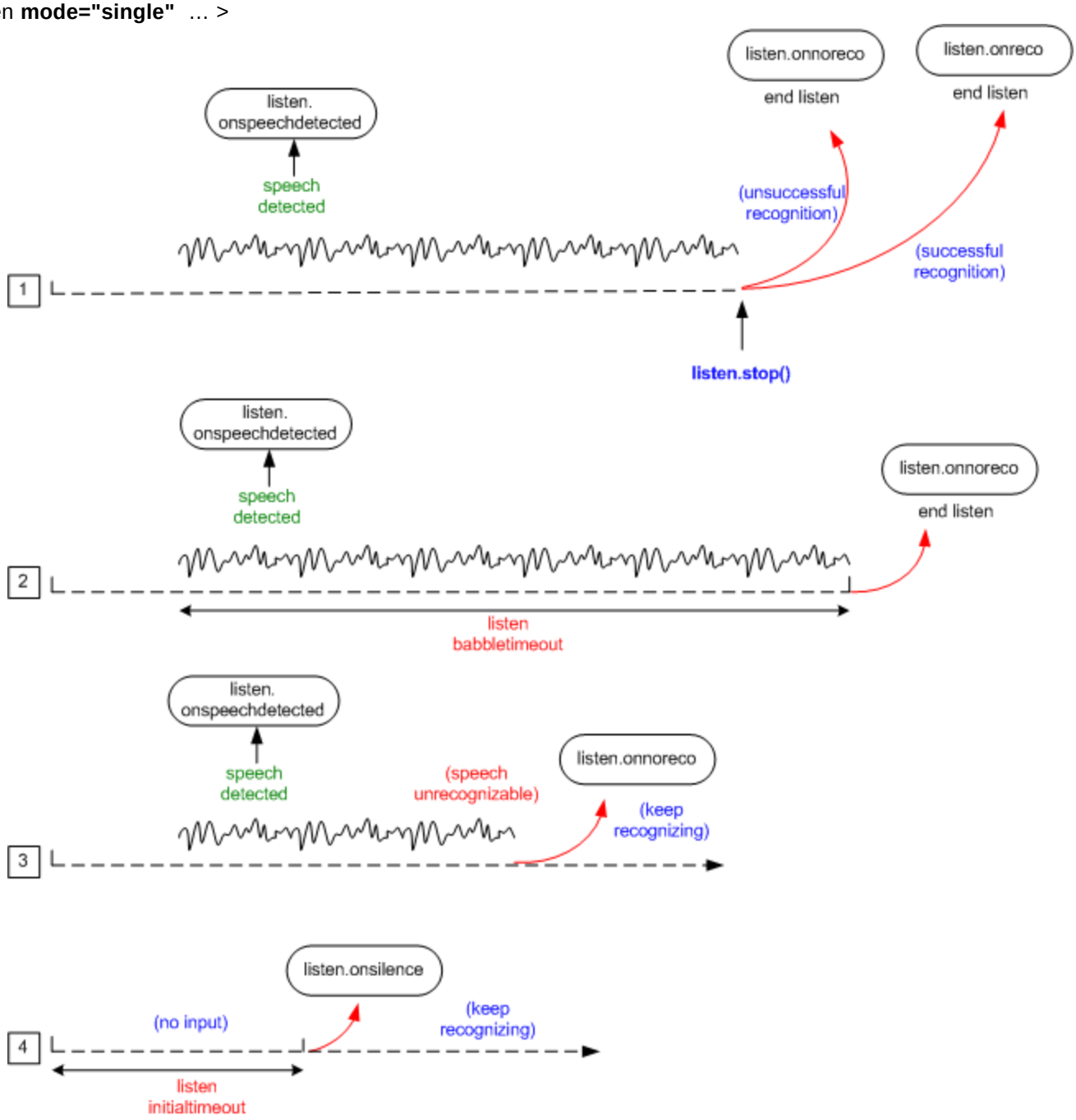


Figure 2: Single mode listen timeline

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 2: Single mode listen timeline shows the common speech recognition events and their behavior for a single mode listen. 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 listen. And case (4) shows how, as with all modes, the

onsilence event is thrown if speech is not detected within the timeout period, but for a single mode listen this does not stop recognition. So for single mode listens, the only speech event which automatically halts execution before a stop call is onnoreco as a result of babbletimeout (along with the non-speech events onerror and onaudiointerrupt).

2.6.3 Multiple mode

<listen **mode="multiple"** … >

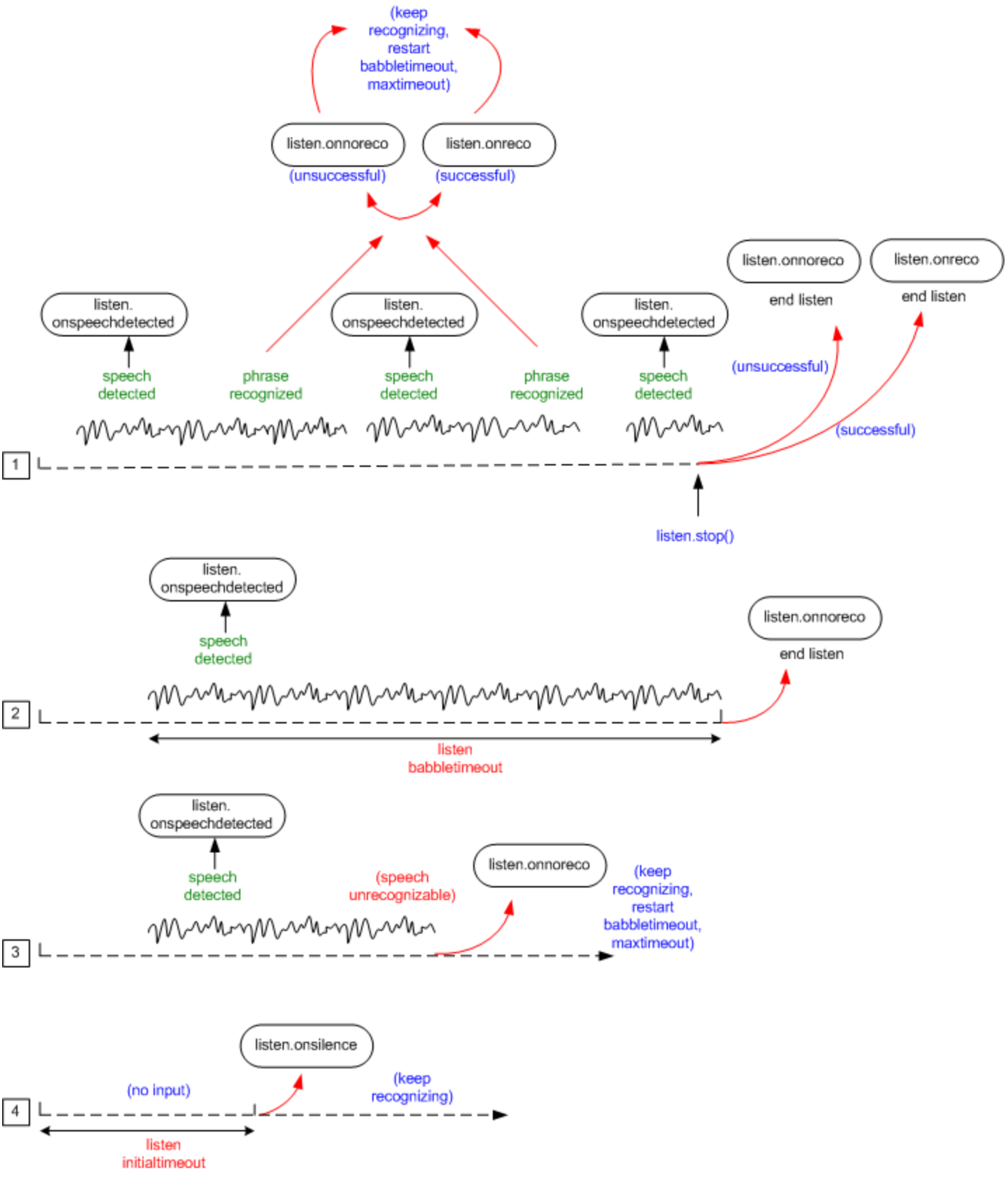


Figure 3: Multiple mode listen timeline

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 phase recognized, an onspeechdetected and an onreco event is thrown and the result returned. This is shown in case (1) of **Figure 3: Multiple mode listen timeline**. 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.

2.7 Events which stop listen execution

The following is a summary of the commands and events 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
- listen.onaudiointerrupt

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

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 8.8.

2.8.1 <listen> content for recording

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

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 2.1.1. (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.

2.8.1.3 <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: mime-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 2.8.2.2). In the case of a totally unsuccessful recording, recordlocation and recordtype will hold empty strings, and recordduration and recordsize will hold values of zero.

2.8.1.4 <param> element

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

2.8.2 Attributes and properties

2.8.2.1 Attributes

The following attributes of <listen> are used to configure the speech recognizer for recording.

- **initialtimeout**: Optional. The time in milliseconds between start of recording 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 2.4.2)[3]. If not specified, the speech platform will use a default value.
- **babbletimeout**: For recording, this sets the time limit on the amount of audio that can be recorded once speech has been detected[4]. If babbletimeout is exceeded, the onnoreco event is thrown but different status codes are possible. If exceeded, the onnoreco event is thrown with status code -15 (see section 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**: this is used in the standard way as for a typical <listen> (see 2.2.1).
- **endsilence**: Optional. For recording, 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**: (this attribute is ignored for listens which perform recording only).
- **xml:lang:** (this attribute is ignored for listens which perform recording only).
- **mode:** Where recognition is enabled along with recording, this is used in the standard way as for a typical <listen> (see 2.2.1). For listens used only for recording, it is ignored.

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 2.8.1.3.

- **recoresult** Read-only. As for recognition, the results of recording in an XML DOM node holding the return document described in 2.8.1.3. For listens used for simultaneous recording and recognition, the return document will hold information for both results.
- **text** Read-only. (Only used when recognition is enabled along with recording).
- **status** Read-only. Status code returned by the recognition platform. Status codes from -20 to -24 are relevant for errors specific to the audio recording process (see 2.8.5.1).
- **recordlocation** Read-only. This holds the location of the recorded audio in a URI.
- **recordtype** Read-only. This holds the mime type of the recorded audio.
- **recordduration**: Read-only. This holds the approximate length of the recording in milliseconds.
- **recordsize**: Read-only. This holds the size of the recorded audio file in bytes.

[3] 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.
[4] 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.

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.

2.8.3.1 Start

This method is used to start audio recording.

Syntax:
Object.Start()
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 2.4.5 for the non-zero status codes.

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 once the platform completes the audio recording.

Syntax:
Object.Stop()
Return value:
None.
Exception:
There are no explicit exceptions associated with this method. However, if recording is successful, onreco is fired, otherwise an onerror event is fired and the status codes as outlined in section 2.8.5.1 are set.

2.8.3.3 Cancel

This method is used to cancel audio recording when the recognition mode is set to 'record'. Any written audio data may be removed by the platform. No events are fired when this method is called.

Syntax:
Object.Cancel()
Return value:
None.
Exception:
None.

2.8.3.4 Activate

This method is used only when recognition is enabled, and is as described in section 2.3.4.

2.8.3.5 Deactivate

This method is used only when recognition is enabled, and is as described in section 2.3.5.

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 2.6.1.). For listens which accomplish recognition along with recording, the mode of recognition used will determine which events are thrown and their behavior (see section 2.6 for different modes of recognition).

2.8.4.1 **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.

Syntax:

Inline HTML	<listen onreco **="***handler***" >**
Event property	Object.onreco = *handler*; Object.onreco = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Recording is accomplished
Default action	Return recognition result object

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).

2.8.4.2 **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 2.8.2.1). This event cancels the audio recording process automatically.

Syntax:

Inline HTML	<listen onsilence="*handler*" …>
Event property (in ECMAScript)	Object.onsilence = *handler* Object.onsilence = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Recognizer did not detect speech within the period specified in the initialtimeout attribute.
Default action	Set status = -11

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

2.8.4.3 **onnoreco**:

For recording-only listens, this event is thrown when the 'babbletimeout' on the recording has been exceeded. In this case the platform also returns the recording results via the 'recoresult' property.

Syntax:

Inline HTML	<listen.onnoreco **="***handler***" >**
Event property	Object.onnoreco = *handler*; Object.onnoreco = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	babbletimeout expires during audio recording.
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 2.2.1).

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

2.8.5 onaudiointerrupt:

The onaudiointerrupt event is fired when the communication channel with the user is lost. In telephony scenarios, for example, it will fire when the line is disconnected. Its default action is to stop recording and return the result up to that point (i.e. just as in listen.stop). Its handler is typically used for local clean-up of resources relevant to an individual listen. In relevant profiles, this event is also fired on <dtmf> and the PromptQueue and call control objects. The order of firing is as follows:

1. listen
2. dtmf
3. PromptQueue object
4. call control object.

Syntax:

Event property	Object.onaudiointerrupt= *handler*; Object.onaudiointerrupt = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The system detects the loss of the connection of the communication channel with the user.
Default action	stop recording and return results.

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

2.8.5.1 **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 info table below.

Syntax:

Inline HTML	<listen onerror **="***handler***" >**
Event property	Object.onerror = *handler*; Object.onerror = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The recording process (once the Start method has been invoked) experiences a serious or fatal problem.
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 both Format and Codec are unsupported one of the status will 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.

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

2.8.6 Stopping audio recording

Audio recording is stopped by any of the means of stopping a listen object in automatic mode (see section 2.7). Since a result is always returned to the listen object, recording listens which are stopped by dtmf input or hang-ups (as in many voice mail applications) always contain recording results if available.

2.9 Advanced speech recognition technology

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

3 Speech output: <prompt >

The prompt element is used to specify the content of audio output. Prompts are queued and played using a prompt queue, which in uplevel browers is exposed to the application as the PromptQueue object (see section 3.5).

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.

In uplevel browsers, prompts are queued individually onto the PromptQueue object (defined in section 3.5) using the Queue method. The PromptQueue object is then used to control the playback of the system. In general the distribution of features between PromptQueue and <prompt> reflects the distinction that the PromptQueue object is used for global audio play settings, and individual prompts are used for local elements of output. In HTML profiles the PromptQueue is a child of the window object (see 10.2.2.1.4).

3.1 prompt content

The prompt element contains the resources for system output, as text or as references to audio files, or both. It also permits platform-specific configuration using the <param> element.

3.1.1 Text and inline tts markup

Simple prompts need specify only the text required for output, eg:

```
<prompt id="Welcome">
   Thank you for calling ACME 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. A SALT browser may support any other speech synthesis formats. Note: 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:

```
<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>
```

3.1.2 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:

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

3.1.3 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 mime-type corresponding to the speech output format used. For XML content, typical types may be the W3C Speech Synthesis Markup Language format, specified as type="application/ssml+xml", or proprietary formats such as "application/x-sapitts+xml". Typical binary content MIME type is "audio/wav" for WAV (RIFF header) 8kHz 8-bit mono mu-law [PCM] single channel or "audio/basic" for Raw (headerless) 8kHz 8-bit mono mu-law [PCM] single channel. (G.711). 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.

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

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

3.1.4 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:

```
<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 may also be defined using the <meta ...> element (see 10.2.2.1.5).

3.2 Attributes and properties

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

3.2.1 Attributes

- **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)[5]. If true the platform will stop the prompt in response to input and flush the prompt queue. 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 3.4.2).
- **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/ for usage.
- **xml:lang:** Optional. String indicating the language of the prompt content. The value of this attribute follows the xml:lang definition in RFC3066. 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 2.1.1). If xml:lang is not specified at any level, the platform is free to determine the language of choice.

3.2.2 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 3.4.1). For each playback, the property is set to null until a bookmark event is encountered.
- **status:** Read-only. 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 as such should be examined in the handler of the relevant event. A status code of zero is set by the oncomplete event (see 3.4.3). Other status values are set when the onerror event is thrown (see 3.4.4).

3.3 prompt methods

Prompt queuing and manipulation may be controlled using the following methods on the prompt object. In this way, uplevel browsers can queue prompt objects.

3.3.1 Queue

Queue the prompt on the PromptQueue object (see section 3.5). 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, it is treated as the string to be output. This argument overrides any inline content, and is

[5] 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.

subject to any relevant features specified in the prompt's attributes in section 3.2.1 (i.e. bargein, xmlns and xml:lang). When a prompt finishes playback the oncomplete event is thrown, and its status code is set to zero.

Syntax:

Object.Queue([strText]);

Parameters:

- **strText:** optional. The text or markup to be sent to the speech output engine. If present, this argument overrides 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 3.1.3), the onerror event is thrown, and the prompt's status code is set to one of the relevant values described in 3.4.4.

Content, whether inline or passed as an argument, must be resolved completely in order for queuing to succeed, so it is on the Queue() call that content is checked for validity and freshness. As noted in section 3.1.3, 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. Prompt playback does not begin until Start() is called (on a prompt or the PromptQueue). If Queue() is called during playback (i.e. after PromptQueue.Start() but before the onempty event is thrown), the prompt is added to the current queue, and will be played back without requiring a further explicit call to start the queue.

3.3.2 Start

Queue the prompt on the PromptQueue object (see section 3.5) and begin playback immediately. 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, it 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 3.2.1 (i.e. bargein, xmlns and xml:lang). This method can be thought of a shorthand for the two commands: prompt.Queue([arg]) followed by PromptQueue.Start, and its content, arguments and the possible resulting events are entirely as for these functions.

Syntax:

Object.Start([strText]);

Parameters:

- **strText:** the text or markup to be sent to the speech output engine. If present, this argument overrides the contents of the object.

Return value:

None.

Exception:

In the event of a queuing problem, the onerror event is thrown, and the prompt's status code is set according to the codes in section 3.4.4. In the case of a PromptQueue object problem after a successful queue, the PromptQueue's status codes are set according to the values in 3.5.3.2.

3.4 prompt events

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

3.4.1 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 8.2.2 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.

Syntax:

Inline HTML	<prompt onbookmark="*handler*" …>
Event property	Object.onbookmark = *handler* Object.onbookmark = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	A bookmark in the rendered string is encountered
Default action	Returns the bookmark string

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

3.4.2 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 2.4.3), or by the onkeypress event or onnoreco event on a started dtmf object for in-grammar and out-of-grammar keypresses respectively (sections 4.4.1, 4.4.3)[6].

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 this event fires and while its associated processing is executed (unless of course it is explicitly stopped elswehere in the script). This may be used in an email reader application, for example, where commands are enabled which do not require the prompt to stop, eg 'speak louder' or 'read faster', or for bookmarking, such as the example in 8.2.2.

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

It should not need restating that whether or not this event is specified, prompt playback is automatically stopped 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 prompt.Stop()).

Syntax:

Inline HTML	<prompt onbargein="*handler*" …>
Event property	Object.onbargein = *handler* Object.onbargein = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	A speech/dtmf input event is encountered
Default action	None

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

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

3.4.3 oncomplete

This event fires when the prompt playback completes normally. It has no effect on the rest of the prompt queue maintained by the PromptQueue object. After the oncomplete event of the last prompt in the queue, the onempty event is thrown by the PromptQueue object (section 3.5.3.1).

Syntax:

Inline HTML	<prompt oncomplete="*handler*" …>
Event property	Object. oncomplete = *handler* Object. oncomplete = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	prompt playback completes
Default action	Set status = 0.

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

3.4.4 onerror

The onerror event is fired if a serious or fatal error occurs with a prompt such that it is unable to be queued. The onerror event will therefore be thrown after the Queue() command, and before playback of the current queue begins. Different types of errors are distinguished by status code and are shown in the event object info table below. The throwing of this event by a single prompt will flush the entire queue maintained by the PromptQueue object.

Syntax:

Inline HTML	<prompt onerror **="***handler***" >**
Event property	Object.onerror = *handler*; Object.onerror = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The synthesis process experiences a serious or fatal problem.
Default action	On encountering an error, status codes are set as follows: **status -1:** Failure to queue the prompt onto the PromptQueue object. **status -2:** Failure to find a speech output resource (for distributed architectures) **status -3:** An illegal property/attribute setting that causes a problem with the synthesis request. **status -4:** Failure to resolve content – this is likely to be an unreachable URI, or malformed markup content.

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

3.5 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. In an XHTML profile, the PromptQueue object will be a child of the Window object (see section 10.2.2.1.4).

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. It 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).

The PromptQueue object contains the following methods and properties.

3.5.1 PromptQueue properties

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

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

3.5.2 PromptQueue methods

3.5.2.1 Start

Start playback of the prompts queued. When the final prompt in the queue finishes playback (and after the throwing of that prompt's oncomplete event), an onempty event is thrown to the PromptQueue object (see 3.5.3.1) and its status property is set to zero. If no prompts are in the prompt queue, or a problem arises with the speech output resource, this call throws an onerror event with the error codes listed in 3.5.3.3.

Syntax:

Object.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 3.5.3.3.

3.5.2.2 Pause

This method pauses playback without flushing the audio buffer. This method has no effect if playback is paused or stopped.

Syntax:

Object.Pause();

Return value:

None.

Exception:

In the event of a problem the onerror event is fired, and the status code is set to -1.

3.5.2.3 Resume

This method resumes playback without flushing the audio buffer. This method has no effect if playback has not been paused.

Syntax:

Object.Resume();

Return value:

None.

Exception:

In the event of a problem the onerror event is fired, and the status code is set to -1.

3.5.2.4 Stop

Stop playback and flush the audio buffer. If playback is not underway (i.e. it has not been started or has already been stopped or paused) the method simply flushes the prompt queue and audio buffer.

Syntax:

Object.Stop();

Return value:

None.

Exception:

None.

3.5.2.5 Change

Change speed and/or volume of playback. Change may be called during playback..

Syntax:

Object.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=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 =0 means to restore the default value.

Return value:

None.

Exception:

None.

3.5.3 PromptQueue event handlers

3.5.3.1 onempty:

This event fires when all prompts queued have finished playback. It fires after the oncomplete is fired on the last individual prompt queued on the object (and therefore only fires when prompt playback completes naturally without explicit stop calls).

Syntax:

Event property	Object.onempty = *handler* Object.onempty = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	All prompts queued on PromptQueue complete playback.
Default action	Set status = 0 if playback completes normally.

Event Properties:

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

3.5.3.2 onaudiointerrupt:

The onaudiointerrupt event is fired when the communication channel with the user is lost. In telephony scenarios, for example, it will fire when the line is disconnected. Its default action is to stop the playback and flush the prompt queue. Its handler is typically used for local clean-up of resources relevant to the prompts of a given page. In relevant profiles, this event is also fired on <listen>, <dtmf> and the call control object. The order of firing is as follows:

1. listen
2. dtmf
3. PromptQueue
4. call control object.

Syntax:

Event property	Object.onaudiointerrupt= *handler*; Object.onaudiointerrupt = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The system detects the loss of the connection of the communication channel with the user.
Default action	Stop prompt playback and flush the prompt queue.

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

3.5.3.3 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 prompt queue is flushed. For platform errors, a status code of -1 is set; if onerror is fired due to start being called on an empty queue, a status code of -2 is set.

Syntax:

Event property	Object.onerror = *handler*; Object.onerror = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The synthesis process experiences a serious or fatal problem.
Default action	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 queue.

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

4 DTMF input : <dtmf>

The <dtmf> element is used in telephony applications to specify possible DTMF inputs and a means of dealing with the collected results and other DMTF events. Like <listen>, its main elements are <grammar> and <bind>, and it holds resources for configuring the DTMF collection process and handling DTMF events. The dtmf element is designed so that type-ahead scenarios are enabled by default. That is, for applications to ignore input entered ahead of time, flushing of the dtmf buffer has to be explicitly authored.

4.1 Content

Mirroring the listen recognition element, the DTMF element holds as content the <grammar> and <bind> elements, and may also be configured in extensible ways with the <param> element.

4.1.1 <grammar>

This is a grammar, as defined in section 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 xml:lang attribute is not meaningful, and within the grammar itself, terminal rules will contain only the digits 0-9, * and # and A, B, C and D as possible tokens. In all other respects, the grammar element is identical to the speech recognition grammar element in section 2.1.1.

4.1.2 <bind>

The bind element is a declarative way to assign the DTMF result to a field in the host page, and is defined in section 2.1.2. <bind> acts on the XML in the result returned by DTMF collection in exactly the same way as it does for listen.

The following example demonstrates how to allow consecutive dtmf input into multiple fields, using remote DTMF grammars and <bind> to update the fields.

```
<xhtml xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
   ...
   <input type="text" name="iptAreaCode" onFocus=”dtmfAreaCode.start()”   />
   <input type="text" name="iptPhoneNumber" />
   ...

   <salt:dtmf id="dtmfAreaCode" onreco="dtmfPhoneNumber.Start()">
      <!-- grammar result will contain "smlAreaCode" node -->
      <salt:grammar src="3digits.gram" />
      <salt:bind value="//smlAreaCode" targetelement="iptAreaCode" />
   </salt:dtmf>

   <salt:dtmf id="dtmfPhoneNumber">
      <!-- grammar result will contain "smlPhoneNumber" node -->
      <salt:grammar src="7digits.gram" />
      <salt:bind value="//smlPhoneNumber" targetelement="iptPhoneNumber" />
   </salt:dtmf>
</xhtml>
```

4.1.3 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:

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

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

Note that page-level parameter settings may also be defined using the <meta ...> element (see 10.2.2.1.5).

4.2 Attributes and properties

4.2.1 Attributes

- **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 keystokes, 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. If unsupported by a platform, or unspecified in the application, the value defaults to that used for interdigittimeout.
- **flush:** 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).

4.2.2 Properties

- **dtmfresult:** Read only. XML node storing 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 2.1.2.
- **text:** Read-only string containing 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.
- **status:** Read-only. Status code returned by the dtmf collector. The status property is only meaningful after status-setting events are thrown by the dtmf object, and as such should be examined in the handler of the relevant event. Possible values are 0 for successful recognition, the failure values -1 to -4 (as defined in the exceptions possible on the Start method (section 4.3.1)), status -11 on the reception of onsilence (see 4.4.4), and status -13 or -16 on an onnoreco (see 4.4.3). (These values reflect corresponding status codes in the listen object of section 2.2.2.)

4.3 Object methods:

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

4.3.1 Start

The Start method starts the dtmf collection process, using as grammars all the top-level rules for active dtmf grammars.

Syntax:
Object.Start();
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 4.4.5 for the non-zero status codes.

4.3.2 Stop

Stop dtmf collection and return results received up to the point when collection was stopped. (Any subsequent key strokes entered by the user, however, will remain in the platform buffer.) If the dtmf object has not been started, this call has no effect.

Syntax:
Object.Stop();
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 4.4.5 are set.

4.3.3 Flush

Flush the DTMF buffer. Flush has no effect if called while the <dtmf> object is started.

Syntax:

Object.Flush();

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 4.4.6 are set.

4.4 Events

Like the listen element, although browsers will update the recoresult property for both successful and unsuccessful dtmf recognitions, this property should only be assumed to hold a valid result 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).

4.4.1 onkeypress[7]

Fires on every pressing of a DTMF key which is legal according to the input grammar (otherwise, an onnoreco event is fired). Immediately prior to the event firing, the value of the key pressed is appended to the string in 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 4.6.

Syntax:

Inline HTML	<DTMF onkeypress="*handler*" …>
Event property	Object.onkeypress = *handler* Object.onkeypress = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Press on the touch-tone telephone key pad
Default action	Appends Object.text with key pressed

Event Properties:

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

4.4.2 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 4.6. 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 the input sequence matches a complete path through the grammar and further input is not possible according to that grammar.
2. when the input sequence matches a complete path through the grammar and further input is still possible according to that grammar, after the period specified in the endsilence attribute. (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.)

Syntax:

Inline HTML	<DTMF onreco="*handler*" …>
Event property	Object.onreco = *handler*

[7] For HTML and XHTML, this overrides the default 'onkeypress' event inherited from the HTML control. Only DTMF keypresses fire this event.

	Object.onreco = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	DTMF recognition is complete.
Default action	Returns dtmf result object.

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

4.4.3 onnoreco

Fires when a key is pressed which is not legal according to the DTMF grammar, or when interdigittimeout is exceeded. This event stops the DTMF object automatically 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 4.6.

Syntax:

Inline HTML	<DTMF onnoreco="*handler*" …>
Event property (in ECMAScript)	Object.onnoreco = *handler* Object.onnoreco = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Illegal key press, or exceeding of interdigittimeout period when input is incomplete.
Default action	Appends Object.text with key pressed, stops dtmf collection. Set status property and update dtmfresult. Status codes are set as follows: **status -13**: out-of-grammar dtmf keypress. **status -16**: interdigittimeout was exceeded.

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

4.4.4 onsilence

onsilence handles the event of no DMTF collected by the platform before the duration of time specified in the initialtimeout attribute (see 4.2.1). This event stops the dtmf object atuomatically.

Syntax:

Inline HTML	<dtmf onsilence="*handler*" …>
Event property (in ECMAScript)	Object.onsilence = *handler* Object.onsilence = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Recognizer did not detect DMTF within the period specified in the initialtimeout attribute.
Default action	Set status = -11

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

4.4.5 onaudiointerrupt

The onaudiointerrupt event is fired when the communication channel with the user is lost. In telephony profiles, for example, it will fire when the line is disconnected. Its default action is to stop collection, update dtmfresult and flush the dtmf buffer. Its

handler is typically used for local clean-up of resources relevant to an individual dtmf element. In relevant profiles, this event is also fired on <listen> and the PromptQueue and call control objects. The order of firing is as follows:

1. listen
2. dtmf
3. PromptQueue
4. call control object.

Syntax:

Event property	Object.onaudiointerrupt= *handler*; Object.onaudiointerrupt = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The system detects the loss of the connection of the communication channel with the user.
Default action	Stop dtmf collection, update dtmfresult and flush the dtmf buffer

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

4.4.6 onerror

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

Syntax:

Inline HTML	<dtmf onerror **="***handler***" >**
Event property	Object.onerror = *handler*; Object.onerror = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	The DTMF collection process experiences a serious or fatal problem.
Default action	On encountering an error, the dtmf object is stopped and status codes are set as follows: **status -1:** A generic platform error occurred during DTMF collection. **status -3:** An illegal property/attribute setting that causes a problem with the DTMF collection request. **status -4:** Failure to find resource – in the case of DTMF this is likely to be a the URI of a DTMF grammar.

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

4.5 DTMF event timeline

The following diagram illustrates typical event possibilities for the dtmf element.

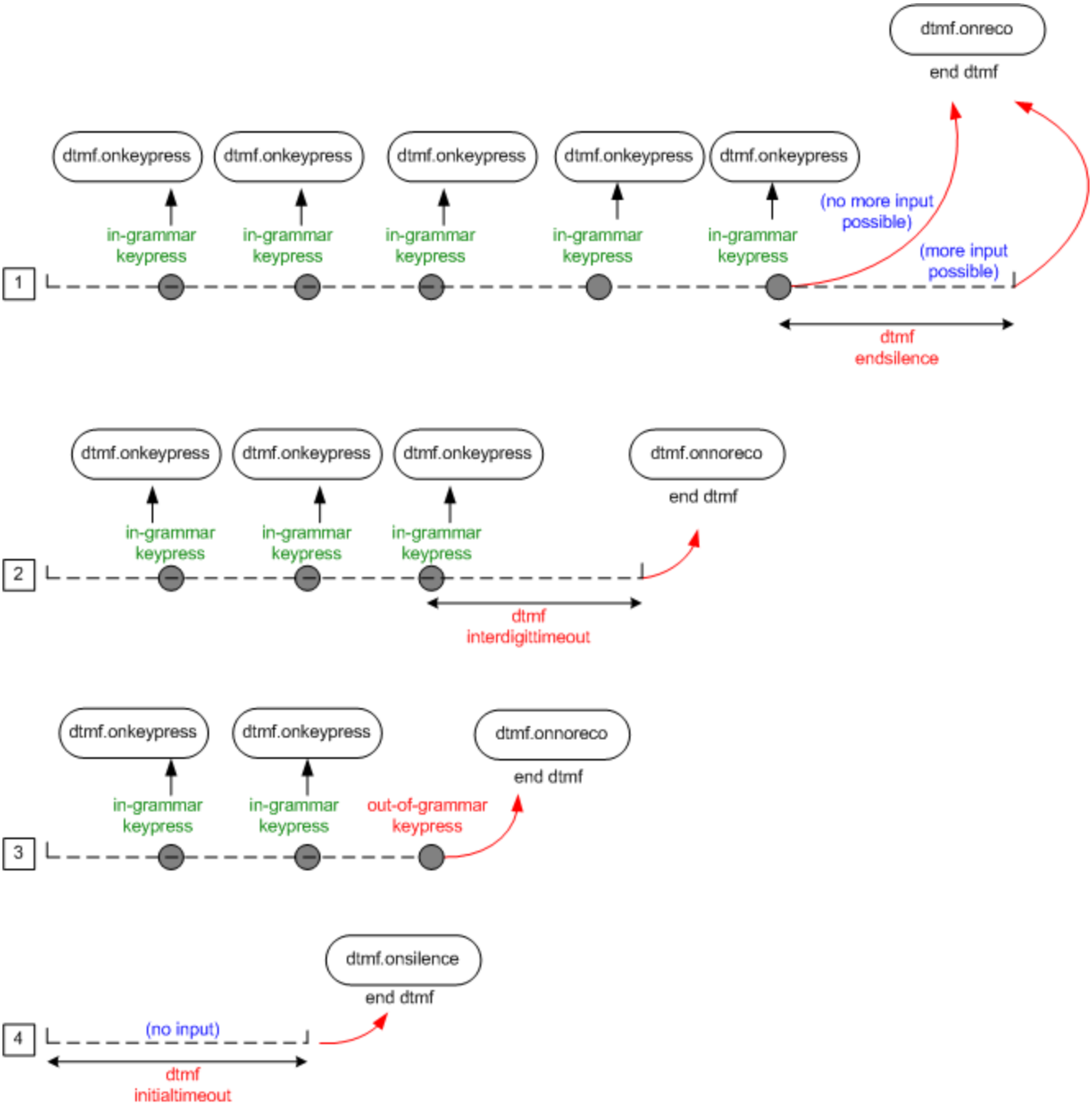


Figure 4: DTMF event timeline

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 (and after the endSilence period if futher 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, and 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.

4.6 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.

4.6.1 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>[8].

```
dmtf.onkeypress         --> disable listen timeouts
listen.onspeechdetected --> disable dtmf timeouts
```

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 initaltimeout 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.

4.6.2 Automatic stop

When one mode stops and throws an event of onsilence, onnoreco or onreco, the other mode is automatically stopped. (Stopping actually occurs before the object receives the event, in order for the event handler functions to operate under a predictable situation.) A result should 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.

```
dmtf.onsilence    --> stop listen
dmtf.onnoreco     --> stop listen
dmtf.onreco       --> stop listen

listen.onsilence  --> stop dtmf
listen.onnoreco   --> stop dtmf
listen.onreco     --> stop dtmf
```

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.onreco); 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.

4.6.3 listen and dtmf interaction event timeline

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

[8] Recall from section **Error! Reference source not found.** 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.

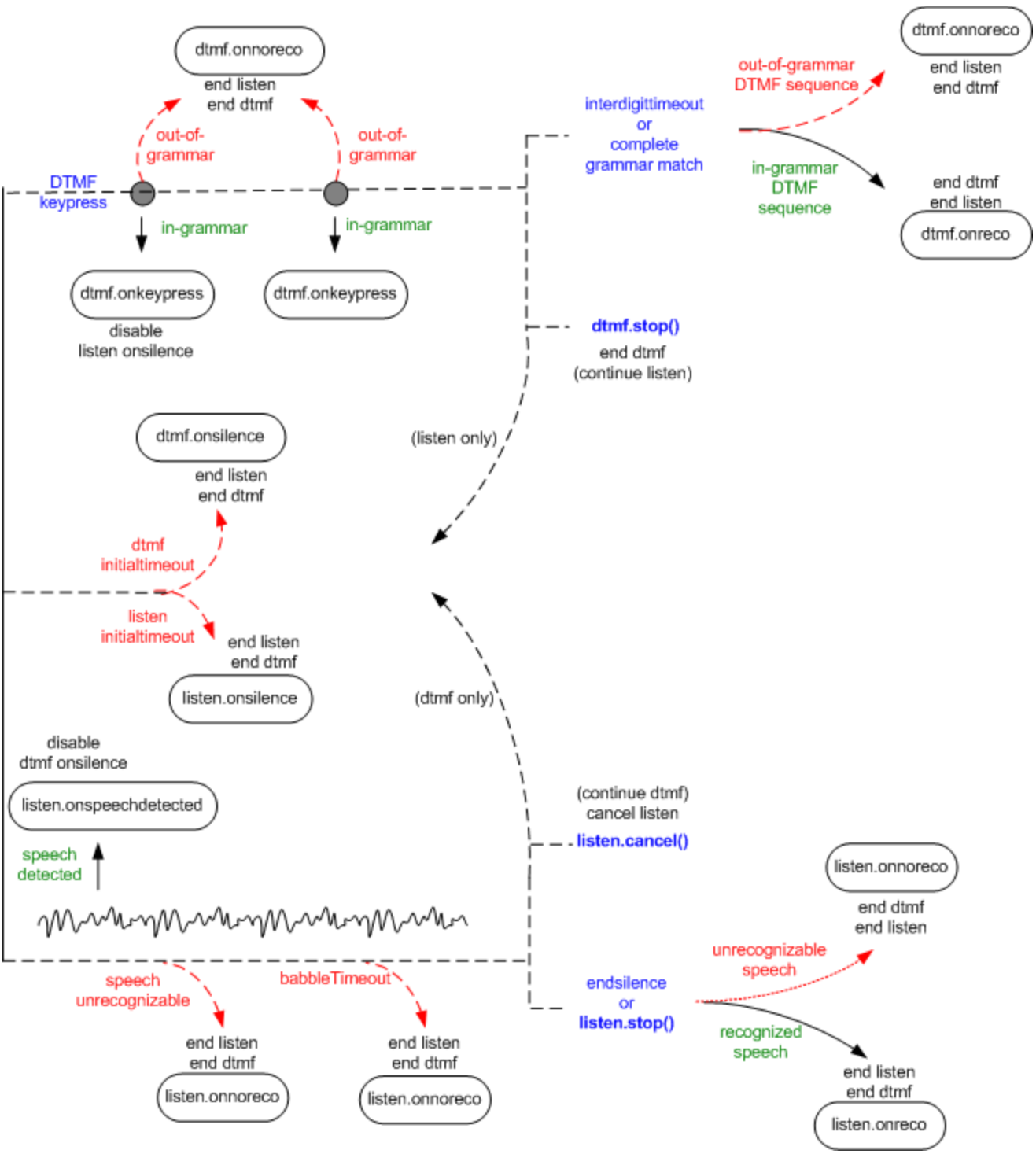


Figure 5: listen and dtmf event interaction

4.7 Events which stop dtmf execution

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

methods

- dmtf.Stop()

dtmf events

- dmtf.onreco

- dtmf.onnoreco
- dtmf.onsilence
- dtmf.onerror
- dtmf.onaudiointerrupt

listen Events

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

5 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 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, i.e. 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 SALT conformance specification may require implementations to support a few core schemas, with strong preferences to existing standard messages formats. 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.

5.1 smex content

smex may have the following child elements:

5.1.1 bind

This is the same element as described in section 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.

5.1.2 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:

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

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

Note that page-level parameter settings may also be defined using the <meta ...> element (see 10.2.2.1.5).

5.2 Attributes and properties

The smex object has the following attributes:

- **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. The value can be changed when a count down is in progress. A zero or negative value stops the clock without triggering the timeout event. The default is 0, meaning no timeout.

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 means normal, timeout expired, and communication with the platform cannot be established or has been interrupted, respectively. Platform specific error messages should conveyed through the received property. For the cases that the error message is successfully delivered, the status code is 0.

5.3 smex events

The object has the following events:

5.3.1 onreceive

The onreceive event is fired when a platform message is received by the browser. 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.

Syntax:

Inline HTML	<smex onreceive **="***handler***" >**
Event property	Object. onreceive = *handler*; Object. onreceive = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Platform message received by the browser.
Default action	Bind directives are evaluated, the *received* property is updated, and *status* code set to zero.

Event Properties:

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

5.3.2 ontimeout

The ontimeout event is fired when the timeout expires.

Syntax:

Inline HTML	<smex ontimeout**="***handler***" >**
Event property	Object.ontimeout=*handler*; Object.ontimeout =GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Time out.
Default action	None

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

5.3.3 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.

Syntax:

Inline HTML	<smex onerror **="***handler***" >**
Event property	Object.onerror = *handler*; Object.onerror = GetRef("*handler*");

Event Object Info:

Bubbles	No
To invoke	Communication error
Default action	Set status codes as follows: **status -1**: communication error

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

6 Telephony Call Control: the CallControl object

This chapter specifies the telephony call control for SALT, the Speech Application Language Tags.

6.1 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).[9] The specified model should support both client and server call control requirements.

6.2 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.

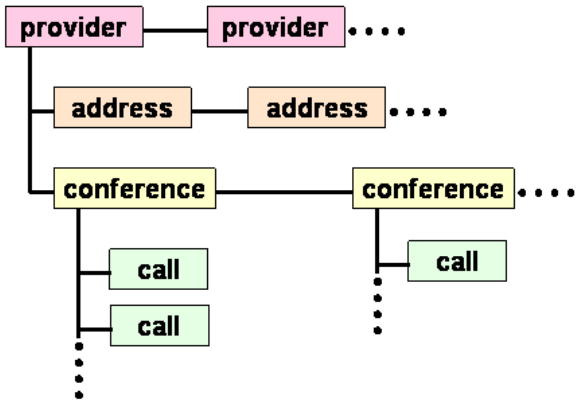
The object shall appear in the DOM of the HTML document. The object will have various properties and methods that can be manipulated via ECMAScript code. Some of these methods will create derivative "child" objects, thereby instantiating an entire call control object hierarchy. The objects should also 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),[10] 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 8.6.

6.2.1 Call Control Object Hierarchy

The SALT call control objects comprise a hierarchy.



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.

[9] 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.

[10] 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.

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

Different telecommunication vendors can market **provider**s 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 **provider**s are present on any given system is a platform browser configuration issue.

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

Providers also allow creation and management of **conference**s, which are logical groupings of **call**s.[11] **Conference**s may be created and terminated. **Call**s may be created and terminated, and also moved into or out of **conference**s. 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).

6.2.2 Browser Configuration of Call Control Providers

The method of instantiation of call control **provider**s 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.

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.

6.2.3 Call Control Event Handling

When the **browser** starts up and each **provider** object is "plugged in", the first event the **provider** throws will be the "`provider.inService`" event.

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

```
<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`

[11] 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.

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.

6.2.4 Lifetime of Objects

Objects such as **conference**s and **call**s 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 **window**s 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 **address**es a **provider** offers; **address**es cannot be created or destroyed, they are essentially a platform provisioning issue.

6.2.5 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 **call**s 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.

6.2.6 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 the platform vendor and inaccessible to the programmer, or 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.

6.3 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.

6.3.1 Events

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

6.3.1.1 Properties

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

6.3.2 callControl Object

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

6.3.2.1 Properties

- `capabilities` – R/O – an XML `documentElement` in a format (to be determined, perhaps Web Services Description Language, WSDL) 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.
- `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 undefined. 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 **call**s 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 undefined. 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 **provider**s configured into the system and accessible through the **browser**.
- `provider.length` – R/O -- number of **provider**s configured into the system

6.3.2.2 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.

6.3.2.3 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.

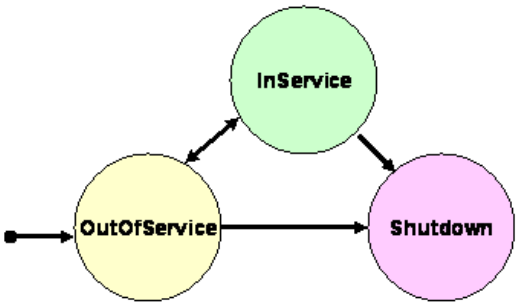
6.3.3 Provider Object

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

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

6.3.3.1 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. **Provider**s 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. 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.

6.3.3.2 Properties

- `address[]` – R/O -- array of **address**es hosted by the **provider**.
- `address.length` – R/O -- number of listenable **address**es hosted by the **provider**.
- `conference[]` – R/O -- array of **conference**s.
- `conference.length` – R/O -- number of child **conference**s 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.

6.3.3.3 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.

6.3.3.4 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.

6.3.4 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, **address**es 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, **address**es 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`").[12]

How an application registers one or more **address**es 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 **address**es are available for use is a **provider** provisioning/configuration issue.

[12] The "`callto:`" URI namespace as used in Microsoft NetMeeting was never formally registered with the IETF and is deprecated by RFC 2806.

6.3.4.1 State Machine

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

6.3.4.2 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.

6.3.4.3 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 **address**es may receive incoming calls, which is useful on platforms (especially servers) that have multiple **address**es. 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 **call**s are automatically accepted (value `True`, the default) so that explicit invocations of `accept()` are not required; or whether incoming **call**s 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.

6.3.4.4 Events

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

6.3.5 Conference Object

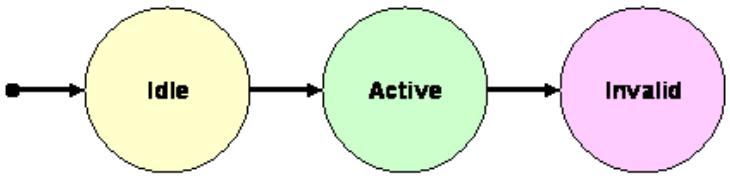
A **conference** is a logical grouping of **call**s that can share their media streams.

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 **call**s 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 **call**s can enter an `Invalid` **conference**. Properties may be examined, and then the **conference** can be destroyed, and a new one created if needed.

6.3.5.1 State Machine



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

- `Active` – A **conference** with some current ongoing activity is in this state. **Conference**s with one or more associated **call**s must be in this state.
- `Idle` -- This is the initial state for all **conference**s, immediately after creation. In this state, the conference has zero **call**s.

- `Invalid` -- This is the final state for all **conference**s. **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. **Conference**s in this state have zero **call**s and these **conference** objects may not be used for any future action.

6.3.5.2 Properties

- `call[]` – R/O -- array of the **call**s in the **conference**.
- `call.length` – R/O -- number of active **call**s 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 – symbolic globally 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.

6.3.5.3 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 **call**s are disconnected, resulting in `call.disconnected` events being thrown.
    - 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 **call**s 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.

6.3.5.4 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**.

6.3.6 Call Object

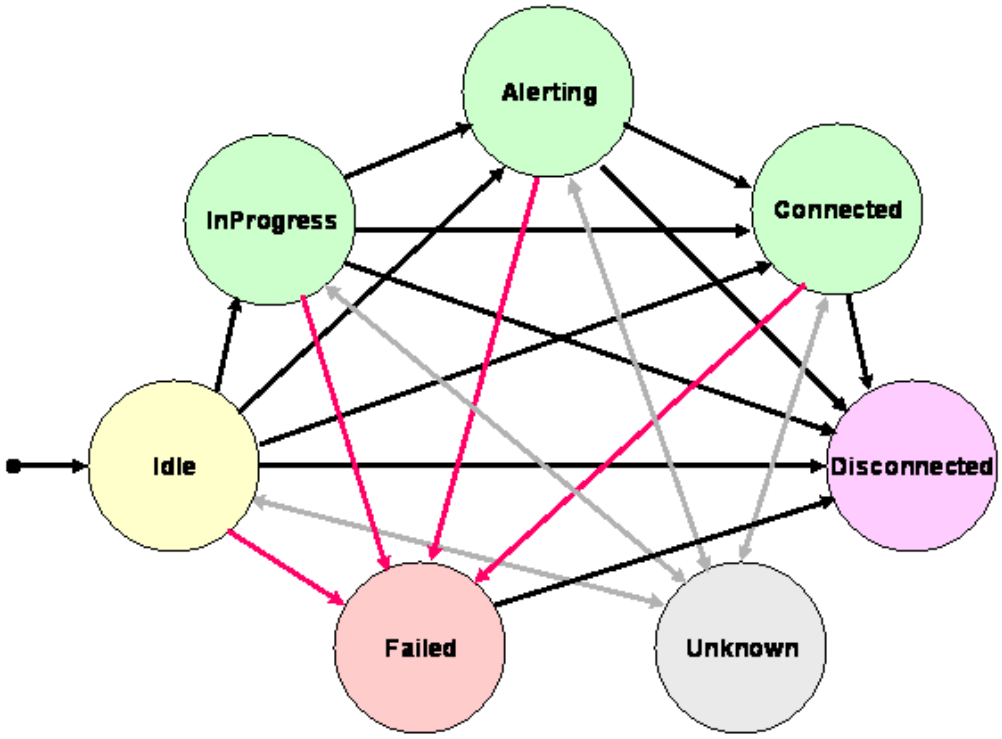
A **call** is a connection between an external endpoint **address** and an endpoint **address** on the platform.

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 **call**s 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.

6.3.6.1 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 **call**s 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 **call**s. **Call**s which are in the `Idle` state are not actively part of a telephone call. **Call**s 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 **call**s that are in this state. **Call**s may move in and out of the `Unknown` state at any time.

6.3.6.2 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.

6.3.6.3 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.[13] 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.

[13] 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.

- `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.

- `disconnect(`*`[reason]`*`)` -- hang up on a **call**. The optional *reason* parameter is a character string describing the reason the call was disconnected. Specific "well known values" for *reason* are to be determined. Note that disconnecting a call will result in a `call.disconnected` event being thrown, and may also result in the `onaudiointerrupt` handler of one or more of the **<listen>**, **<dtmf>** or PromptQueue objects being invoked.

- `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. Specific "well known values" for *reason* are to be determined.

- `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`). 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.

6.3.6.4 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`.

Value	***Description***
`presentation-allowed`	Display URI and name.
`presentation-restricted`	Do not display URI and name: show "private".
`number-lost-due-to-interworking`	Information not available for display: show "unknown" or "out of area".
`reserved-value`	Implementation specific.

- 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`.

Value	***Description***
`user-provided-unscreened`	The application set URI and name, and has not screened it.
`user-provided-passed`	The application set URI and name, has screened it, and it passed screening.
`user-provided-screening-failed`	The application set URI and name, has screened it, and it failed screening.
`network-provided`	The network set URI and name.

6.3.6.5 Events

- `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 (e.g., far-end hang up, credit expired, time exceeded, etc.) 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 (e.g., busy, no answer, network congestion, etc.).
- `call.inProgress` -- an outbound call is in the process of connecting to the remote end.
- `call.unknown` – the **call** is in an unknown state.

7 Logging

This section is under development. It will define a means to enable general platform logging using a global script function.

8 SALT illustrative examples

8.1 Controlling dialog flow

8.1.1 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 xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
      ...
      <input id="txtBoxCity" type="text" />
      <input id="buttonCityListen" type="button" onClick="listenCity.Start();" />
      ...

      <!—- SALT -->
      <salt:listen id="listenCity">
          <salt:grammar name="g_city" src="./city.xml" />
          <salt:bind    targetelement="txtBoxCity"
                        value="//city" />
      </salt:listen>
</html>
```

8.1.2 Using HTML and script to implement dialog flow

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 xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
      <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);
                  } else {
                        // proceed with value assignments
                  }
            }
            ]]>
            </script>
      </head>
<body onload="RunSpeech()">
      <salt:prompt id="p_help" oncomplete=" RunSpeech()" />
      <salt:prompt id="p_stock" oncomplete="g_stock.Start()">
            Please say the stock name
      </salt:prompt>
      <salt:prompt id="p_op" oncomplete="g_op.Start()">
            Do you want to buy or sell
      </salt:prompt>
      <salt:prompt id="p_quantity" oncomplete="g_quantity.Start()">
            How many shares?
      </prompt>
      <salt:prompt id="p_price" oncomplete="g_price.Start()">
            What's the price
      </salt:prompt>

      <salt:listen id="g_stock" onreco="handle(); RunSpeech()" >
            <salt:grammar src="./g_stock.xml" />
      </salt:listen >

      <salt:listen id="g_op" onreco="handle(); RunSpeech()" />
            <salt:grammar src="./g_op.xml" />
      </salt:listen >

      <salt:listen id="g_quantity" onreco="handle(); RunSpeech()" />
            <salt:grammar src="./g_quant.xml" />
      </salt:listen >

      <salt:listen id="g_price" onreco="handle();RunSpeech()" />
            <salt:grammar src="./g_quant.xml" />
      </salt:listen >

      <form id="trade">
            <input name="stock" title="stock name" />
```

```
                <select name="op" title="buy or sell">
                      <option value="buy" />
                      <option value="sell" />
                </select>
                <input name="quantity" title="number of shares" />
                <input name="price" title="price" />
          </form>
      </body>
</html>
```

8.1.3 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 data and UI is separated, the app developers only need to change the speech section when changing interaction style to mixed initiative. The data section remains the same.

Here is briefly how it works:

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 everything is okay, the form is submitted, all without scripting.

```
<html xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
      <body>
            <!—-  the data section -->
              <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” />

                  <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” />

                <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:bind test=”/Call_connected”
                           targetelement=”welcome” targetmethod=”queue” />
                <salt:bind test=”/Call_connected”
                           targetelement=”ask” targetmethod=”start” />
                <salt:bind test=”/Call_connected”
                           targetelement=”listen_team” targetmethod=”start” />
          </salt:smex>
      </body>
</html>
```

8.2 Prompt examples

8.2.1 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 barge-in 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 xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
  <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>
```

8.2.2 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 when bargein happened during the prompt output. The onbargein handler calls a script which sets a global ‘mark’ variable to the last bookmark encountered in the prompt, and the value of this ‘mark’ is used in the listen’s postprocessing function (‘ProcessCityConfirm’) to set the correct value.

```
     <script><![CDATA[
           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;
           }
     ]]></script>
     <body xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
     <input name="txtBoxOrigin" value="Seattle" type="text" />
     <input name="txtBoxDest" type="text" />
     ...
```

```
      <salt:prompt id="confirm" onbargein="interrupt()" bargein="true">
            From <bookmark mark="mark_origin_city" />
            <value targetelement="txtBoxOrigin" targetattribute="value" />,
            please say <bookmark mark="mark_dest_city" /> the
            destination city you want to travel to.
      </salt:prompt>
      <salt:listen onreco="ProcessCityConfirm()" >
            <salt:grammar src="/grm/1033/cities.xml" />
      </salt:listen>
      …
      </body>
```

8.3 Using SMIL

The following example shows activation of prompt and listen elements using SMIL mechanisms.

```
      <html xmlns:t="urn:schemas-microsoft-com:time"
            xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
            <head>
                  <style>
                    .time { behavior: url(#default#time2); }
                  </style>
            </head>
            <body>

                  <input name="txtBoxOrigin" type="text" />
                  <input name="txtBoxDest" type="text" />

                  <salt:prompt class="time" t:begin="0">
                      Please say the origin and destination cities
                  </salt:prompt>

                  <t:par t:begin="time.end" t:repeatCount="indefinitely"
                    <salt:listen class="time">
                      <salt:grammar src="./city.xml" />
                      <salt:bind targetelement="txtBoxOrigin"
                                 value="//origin_city" />
                      <salt:bind targetelement="txtBoxDest"
                                 test="//dest_city[@confidence &gt; 40]"
                                 value="//dest_city" />
                    </salt:listen>
                  </t:par>

            </body>
      </html>
```

8.4 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 xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">

<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.xml" />
      </salt:listen>

      <salt:listen id="recoDestCity"
                  onreco="procDestCity()"
                  onnoreco=”procNothingUnderstood()”
                  onsilence=”procNothingUnderstood()”
                  onerror=”procError()”>
            <salt:grammar src="./city.xml" />
      </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();
                  }
            }
            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();
            }

      </script>

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

</body>
</html>
```

8.5 smex examples

8.5.1 logging

```
<salt:smex id="logServer">
      <salt:param name="d:server" xmlns:d="urn:Microsoft.com/COM">
            <d:protocol>DCOM</d:protocol>
            <d:clsid>20930930293020293209420984320 98</d:clsid>
            <d:iid>09038593049034985309853090948 03</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 logging mechanism can be achieved 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 app 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 communicated with the same platform component as long as there won't 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 applies to TCP servers listening to multiple ports.

8.6 Call Control use case examples

8.6.1 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 xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
<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);
              }
       }
]]></script>
</head>
<body>
       <salt:prompt id="askMailbox">
              Welcome, please say your mailbox number.
       </salt:prompt>
       <salt:listen id="recoMailbox" onreco="javascript:procMailbox()">
              <salt:grammar src="./digits.xml" />
       </salt:listen>
</body>
</html>
```

8.6.2 Notification call

In this example, a notification service dials an outbound call to a subscriber to notify him of a pending dentist appointment.

```
<html xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
<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>
```

8.6.3 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"><![CDATA[
	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>
```

8.6.4 Voice Activated Dialing

In this example, a subscriber calls a voice activated dialing service, speaks the number he wants called, and the service places the call using network transfer facilities.

```
<html xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
<head>
<title>Voice Activated Dialing - Intelligent Network Transfer</title>
<script type="text/javascript"><![CDATA[
	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.xml" />
	</salt:listen>
</body>
</html>
```

8.6.5 Voice Activated Dialing with Active Listen

This is a slightly different example of voice activated dialing. The subscriber calls the service, speaks the number he wants connected, and is connected using a “trombone” (or “hairpin”) of the two call legs in a single conference.

```
<html xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
<head>
<title>Voice Activated Dialing - Trombone Conference with Active Listen</title>
<script type="text/javascript"><![CDATA[
	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.xml" />
	</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.xml" />
	</salt:listen>
	<salt:dtmf id="dtmfPoundPound" onreco="javascript:procHangup(callee)">
		<salt:grammar> # # </salt:grammar>
```

```
	</salt:dtmf>
</body>
</html>
```

8.6.6 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 xmlns:salt="urn:schemas.saltforum.org/2002/02/SALT">
<head>
<title>Find Me - Simultaneously Dial Several Numbers</title>
<script type="text/javascript"><![CDATA[
	var caller, callee[3], answerer;
	var phoneNumber[3];
	phoneNumber[0] = "tel:+1-408-555-1212";
	phoneNumber[1] = "tel:+1-415-555-1212";
	phoneNumber[2] = "tel:+1-925-555-1212";
	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++) {
			callee[i] = caller.parent.createCall();
			callee[i].attachEvent("call.connected",
				procCalleeConnected);
			callee[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 != callee[i]) {
					callee[i].disconnect();
				}
			}
		}
	}
	function procTimeout()	{
		promptQueue.stop();
		callControl.mediaDest = caller.channel[0];
		// disconnect all outgoing calls
		for (var i = 0; i < phoneNumber.length; i++) {
			callee[i].disconnect();
		}
		sayNotAvailable.start();
	}
]]></script>
</head>
```

```
<body>
	<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.xml" />
	</salt:listen>
	<salt:prompt id="sayNotAvailable" oncomplete="javascript:caller.disconnect()">
		Sorry, he is not available. Goodbye.
	</salt:prompt>
</body>
</html>
```

8.7 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.

It is also possible to create static web-pages that work equally well with both legacy browsers and multimodal browsers. Because the SALT tags are not recognized by legacy browsers, 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>
	                <rule toplevel="active">
	                    <p>from </p>
	                    <ruleref name="cities" />
	                </rule>
	                <rule name="cities">
	                    <list>
	                          <p> Chicago </p>
	                          <p> Milwaukee </p>
	                          <p> Kalamazoo </p>
	                    </list>
	                </rule>
	            </grammar>
	      </salt:grammar>
	</span>
```

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:

```
	function procOriginCity () {
	      if (txtBoxOriginCity && recoOriginCity) {
```

```
                txtBoxOriginCity.value = recoOriginCity.value;
                RunAsk();
          }
     }
```

8.8 Audio recording example

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

```
<!—- HTML -->
<!—- on page load -->
<body onload=”RunAsk()”>

     <form id=”f1” action=”http://acme.com/savewaveform.aspx” method=”post”>
          <input name="vmail" type="file" />
     </form>

     <!—- Prompts -->
     <salt:prompt id=”p_record” oncomplete=”r_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=”r_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=”r_save” onreco=”processSave()”>
         <salt:grammar src="./yesno.xml" />
     </salt:listen>

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

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

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

     // If user is satisfied post recording back to web server
```

```
      // otherwise start again
      function processSave () {
            smlResult = event.srcElement.recoresult;

            origNode = smlResult.selectSingleNode("//answer/text()");
            if (origNode.value == "Yes") {
                  vmail.value = r_recordvm.recordlocation;
                  f1.submit();
            } else {
                  RunAsk();
            }
      }
      </script>
</body>
```

9 Appendix A: SALT DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for SALT WD 2.0 of 13 Feb 2002 -->

<!ENTITY % boolean "(true | false)">
<!ENTITY % confidence.value "CDATA">  <!-- should be a float between 0.0 and 1.0 -->
<!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
      targetmethod            %object.method;   #IMPLIED
      test                          %xpattern.string; #IMPLIED
      value                   %xpath.query;           #REQUIRED
>

<!ELEMENT content (#PCDATA)>
<!ATTLIST content
      href                    %uri;                   #REQUIRED
      type                    %content.type;          "application/ssml+xml"
>

<!ELEMENT dtmf (grammar | bind | param)* >
<!ATTLIST dtmf
      id                            ID                                #IMPLIED
      endsilence              %milliseconds;          #IMPLIED
      flush                   %boolean;               "false"
      initialtimeout                %milliseconds;          #IMPLIED
      interdigittimeout       %milliseconds;          #IMPLIED
      onaudiointerrupt        %script.statement;      #IMPLIED
```

```
      onerror                   %script.statement;      #IMPLIED
      onkeypress          %script.statement;      #IMPLIED
      onnoreco            %script.statement;      #IMPLIED
      onreco                    %script.statement;      #IMPLIED
      onsilence           %script.statement;      #IMPLIED
>

<!ELEMENT grammar ANY>
<!ATTLIST grammar
      name                CDATA                   #IMPLIED
      src                       %uri;                   #IMPLIED
      type                %content.type;          "application/grammar+xml"
      xmlns               %uri;                   #IMPLIED
      xml:lang                  CDATA                   "en-US"
>

<!ELEMENT listen (record | grammar | bind | param)* >
<!ATTLIST listen
      id                        ID                              #IMPLIED
      initialtimeout            %milliseconds;          #IMPLIED
      babbletimeout       %milliseconds;          #IMPLIED
      maxtimeout          %milliseconds;          #IMPLIED
      endsilence          %milliseconds;          #IMPLIED
      reject                    %confidence.value;      #IMPLIED
      xml:lang                  CDATA                   "en-US"
      mode                %listen.mode;           "automatic"
      accesskey           CDATA                   "#"
      style               CDATA                   "visibility: hidden"
      onaudiointerrupt    %script.statement;      #IMPLIED
      onerror                   %script.statement;      #IMPLIED
      onnoreco            %script.statement;      #IMPLIED
      onreco                    %script.statement;      #IMPLIED
      onsilence           %script.statement;      #IMPLIED
>
<!-- NOTE: accesskey and style attributes are used only in HTML profiles -->

<!ELEMENT param ANY>
<!ATTLIST param
      xmlns               %uri;             #IMPLIED
      name                CDATA             #REQUIRED
>

<!ELEMENT prompt (#PCDATA | content | value | param)* >
<!ATTLIST prompt
      id                        ID                              #IMPLIED
      bargein                   %boolean;               "true"
      prefetch            %boolean;               "false"
      xmlns               %uri;                   #IMPLIED
      xml:lang                  CDATA                   "en-US"
      onaudiointerrupt    %script.statement;      #IMPLIED
      onbargein           %script.statement;      #IMPLIED
      onbookmark          %script.statement;      #IMPLIED
      oncomplete          %script.statement;      #IMPLIED
      onerror                   %script.statement;      #IMPLIED
>

<!-- default audio MIME type conforms to RFC 2361 -->
```

```
<!ELEMENT record EMPTY>
<!ATTLIST record
      type                    %content.type;          "audio/wav;codec=g711"
      beep                    %boolean;               "false"
>

<!ELEMENT smex (bind | param)* >
<!ATTLIST smex
      id                            ID                              #IMPLIED
      sent                    CDATA                   #IMPLIED
      timer                   %milliseconds;          #IMPLIED
      onerror                       %script.statement;      #IMPLIED
      onreceive               %script.statement;      #IMPLIED
>

<!ELEMENT value EMPTY>
<!ATTLIST value
      targetattribute         %script.variable; "value"
      targetelement           %script.variable; #IMPLIED
>
```

10 Appendix B: SALT modularization and profiles

10.1 Modularization of SALT

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

We envisage SALT browsers falling into the following 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. Examples include PDA, smart phones, set top boxes, and some automobile navigation systems, etc.
- Rich Clients: computing devices that rival PCs. Usually, the devices have suitable displays, and UI may be more biased towards a visual design other than for the hands free, eyes free applications. Speech related computations may still be distributed, but a network connection is not mandatory. Examples include desktop, wall, and pocket PCs and some automobile PCs. Rick Clients should have no problem supporting scripting.
- Telephony Servers: SALT browsers are running on server-grade computers that process multiple phone calls. The UI includes speech and DTMF. As of the conference call on Feb. 8, 2002, the group feel requiring scripting support is reasonable.

Many functional features in SALT do not make sense to all the environments. The purpose of SALT modularization is to classify them into proper categories so that browser implementers have the greatest flexibility and the app developers can enjoy maximum interoperability.

10.1.1 Declarative Programming Module

The module contains the <bind> subelement and all its attributes.

Note that app 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.

10.1.2 Basic Recognition Module

The module contains the <listen> object, all the recognition related properties (e.g., text, recoResult), the mode attribute, the <grammar> and <param> subelements and all their attributes, and all the events and methods defined in Chapter 2.

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

10.1.3 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 Chapter 2.

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.

10.1.4 Concurrent Recording and Recognition Module

When a browser claims to support both the basic recognition and the basic recording modules, it 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.

Apps 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.

10.1.5 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, but not text nodes in speech synthesis markup languages.

10.1.6 Speech Synthesis Module

The module contains the <prompt> element and all its properties, methods, events, and subelements. Note that prompt queue object is not included here. This is because a multimedia enabled browser (e.g., SMIL aware) often has more sophisticated mechanism 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.

10.1.7 Messaging Module

This module contains the <smex> element and all its properties, methods, events, and the <param> subelement.

10.1.8 Call Control Module

This module contains the call control object and all its child objects. The module makes sense for smart phones where the call control functions are mostly intended for speech communications with another party. Supporting this module implies the browser also supports the scripting module. However, basic telephony module can be supported in the absence of other speech modules. Therefore, the global logMessage function and the PromptQueue object do not belong here.

10.1.9 DTMF Module

This module contains the DTMF element and all its properties, methods, and events.

10.1.10 PromptQueue Module

The module supports the PromptQueue object and all its properties, events, and methods.

10.1.11 Logging Module

The module contains the global logging function.

10.1.12 Deployment Matrix

This section is under development. It will incorporate a table which maps different classes of client device to each of the SALT modules defined above, in terms of whether support of the module should be mandatory, optional or not applicable to that device.

10.2 SALT/HTML profiles

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

10.2.1 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.

10.2.1.1accesskey 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 capabilites into the <listen> element. 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 for a listen object:

- for 'automatic' mode, the onkeypress event for the accesskey invokes the start method.
- 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".
- for 'continuous' and 'record' modes, the onkeypress event toggles the start and stop methods. In other words, the access key enables "click to talk".

This permits simple declarative statements such as:

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

where the onkeypress event from the "*" will have the behavior above (depending on the mode of listen) that would otherwise need to be scripted directly.

style
When 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.

10.2.2 HTML voice-only

10.2.2.1HTML 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[14].

10.2.2.1.1 XHTML Modules

The following XHTML modules as defined at 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 for finer detail on the level of support for each element).

Required	Module Name	Supported elements

[14] 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.

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

10.2.2.1.1.1 Elements

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

- <!DOCTYPE>
- <html>

- <head>
- <body>
- <title>
-
- <a>
- <form>
- <input>
- <select>
- <option>
- <textarea>
- <meta>
- <script>
- <link>
- <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.

ID	Element	Subcategory	Detail	Comments
General				
1	**<!DOCTYPE>**	**-**	**-**	This tag tells the browser which DTD specification the document uses for validation.
Structure				
2	**body**	**Attribute set**	**Common**	The **body** element defines the documents' body. It contains all the contents of the document.
3	**head**	**Attribute set**	**I18N**	The **head** element can contain information about the document. The following tags can be in the head-section: <base>, <link>, <meta>, <script>, (<style>) and <title>.
	head	**Attribute**	**profile**	A space separated list of URLs. The pages referenced by the list contain meta data information about the page.
4	**html**	**Attribute set**	**I18N**	**Root node for the XHTML document**
5		Attribute	version	
6		**Attribute**	**xmlns**	This attribute defaults to "http://www.w3.org/ 1999/xhtml".The browser checks that this namespace is set.
7	**title**	**Attribute**	**I18N**	This element defines the **title** of the document. (No SALT behavior is directly associated with this element[15].)
Text				
8	abbr	Attribute	Common	
9	acronym	Attribute	Common	

[15] Of course the information held by structural elements can be used indirectly in SALT, as in:

```
<title id="welcomeTitle">
       Welcome to my truly fabulous application.
</title>

<salt:prompt id="welcomePrompt">
       <salt:value targetelement="welcomeTitle" />
</salt:prompt>
```

10	address	Attribute	Common	
11	blockquote	Attribute	Common	
12	blockquote	Attribute	Common	
13	br	Attribute	Common	
14	cite	Attribute	Common	
15	code	Attribute	Common	
16	dfn	Attribute	Common	
17	**div**	**Attribute**	**Common**	The **div** element defines the start of a division/section in a document.
18	em	Attribute	Common	
19	h1	Attribute	Common	
20	h2	Attribute	Common	
21	h3	Attribute	Common	
22	h4	Attribute	Common	
23	h5	Attribute	Common	
24	h6	Attribute	Common	
25	kbd	Attribute	Common	
26	p	Attribute	Common	
27	pre	Attribute	Common	
28	pre	Attribute	Common	
29	samp	Attribute	Common	
30	span	Attribute	Common	
31	strong	Attribute	Common	
32	var	Attribute	common	
Hypertext				
33	**a**	**Attribute**	**Common**	The **anchor** tag defines an anchor element. An anchor can be used in two ways: To create a link to another document by using the href attribute or to create an anchor in a document, by using the id attribute.
34		Attribute	accesskey	
35		**Attribute**	**charset**	Sets the character encoding of the page where the link points
36		**Attribute**	**href**	Creates a link to a document
37		**Attribute**	**hreflang**	Set base language of the page where the link points.
38		**Attribute**	**rel**	Specifies a forward link: alternate, designates, stylesheet, start, next, prev, contents, index, glossary, copyright, chapter, section, subsection, appendix, help, bookmark
39		**Attribute**	**rev**	Specifies a backwards link: alternate, designates, stylesheet, start, next, prev, contents, index, glossary, copyright, chapter, section, subsection, appendix, help, bookmark
40		Attribute	tabindex	
41		**Attribute**	**type**	Gives a hint about the content type of the content where the link points to

42		***DOM method***	***click()***	*This allows a script author to simulate a mouse click. This is not strictly part of the HTML DOM spec (but it is supported in IE).*
Forms				
49	**form**	**Attribute**	**Common**	The **form** element creates a form for user input. A form can contain textfields, checkboxes, radio-buttons and more. With forms the user can pass data to the server. The <form> tag can of course be used without any attributes, but most forms requires the action attribute to do something meaningful. See http://www.w3.org/TR/REC-html40/interact/forms.html#h-17.13 for HTML4 forms.
50		**Attribute**	**accept**	A comma separated list of mime types that the server accepts. A browser can use this information.
51		**Attribute**	**accept-charset**	A comma separated list of character encodings the server must be able to process when the form is submitted. The default value is "unknown"
52		**Attribute**	**action**	URI - Specifies where to send the data when the user pushes the submit button in a form (required attribute)
53		**Attribute**	**method**	The HTTP method of passing the data to the web server, ("get" | "post")
54		**Attribute**	**enctype**	The encoding is only applicable to the "post" method, since "get" is ASCII only.
55		***DOM Property***	***elements***	*Read-only property returns a collection of all form control elements in the form.*
56		***DOM Property***	***length***	*Read only property returns the number of form controls in the form*
57		***DOM method***	***reset()***	*Restores a form element's default values. It performs the same action as a reset button*
58		***DOM method***	***submit()***	*Submits the form. It performs the same action as a submit button.*
59	**input**	**Attribute**	**Common**	The **<input>** tag defines the start of an input field where the user can enter data
60		**Attribute**	**accept**	The different kind of files allowed. Used with type="file"
61		Attribute	accesskey	
62		Attribute	alt	
63		Attribute	checked	
64		Attribute	disabled	
65		Attribute	maxlength	
66		**Attribute**	**name**	This is of type CDATA and is the form control name or object name when submitted with a form. The name attribute can be used in conjunction with a valid reference to the owning form to reference the corresponding form control (or accessed independently via the id attribute) from within script.
67		Attribute	readonly	

68		Attribute	size	
69		Attribute	src	
70		Attribute	tabindex	
71		**Attribute**	**type**	Only ("text" | "submit" | "reset" | "file" | "hidden") types are supported in M2. This leaves "password, checkbox, button, radio, image.
72		**Attribute**	**value**	When the type attribute of the element has the value "text", "file" or "password", this represents the current contents of the corresponding form control, in an interactive user agent. Changing this attribute changes the contents of the form control, but does not change the value of the HTML value attribute of the element. When the type attribute of the element has the value "hidden", "submit", "reset" this represents the HTML value attribute of the element.
73		***DOM Property***	***defaultValue***	*When the type attribute of the element has the value "text", "file" (or "password"), this represents the HTML value attribute of the element. The value of this attribute does not change if the contents of the corresponding form control, in an interactive user agent, changes.*
74		***DOM Property***	***form***	*Returns the FORM element containing this control. Returns null if this control is not within the context of a form.*
75		***DOM method***	***click()***	*This DOM method simulates a mouse click for the <input> element when the type is submit or reset (and although not implemented this also applies to radio, checkbox and button).*
76	**select**	**Attribute**	**Common**	The **select** element creates a drop-down box. Note that the browser should return the first option in a select as the value to pass to the server if no options are specified.
77		Attribute	disabled	
78		**Attribute**	**multiple**	Sets whether multiple items can be selected. Set multiple="multiple" to have multiple items, otherwise do not specify.
79		**Attribute**	**name**	This is of type CDATA and is the form control name or object name when submitted with a form. The name attribute can be used in conjunction with a valid reference to the owning form to reference the corresponding form control (or accessed independently via the id attribute) from within script.
80		Attribute	size	
81		Attribute	tabindex	
82		***DOM Property***	***form***	*Returns the FORM element containing this control. Returns null if this control is not within the context of a form.*
83		***DOM Property***	***length***	*This read-only property returns the number of options in this SELECT*
84		***DOM Property***	***options***	*This read-only property returns the collection of OPTION elements contained by this element*
85	**option**	**Attribute**	**Common**	The **option** element defines an option in the drop-down box.
86		Attribute	disabled	
87		Attribute	label	

88		**Attribute**	**selected**	To select this option the value should be set to "selected".
89		**Attribute**	**value**	Sets the value of the option
90		***DOM Property***	***form***	*Returns the FORM element containing this control. Returns null if this control is not within the context of a form.*
91		***DOM Property***	***index***	*This read-only property returns the index of this OPTION in its parent SELECT, starting from 0*
92	**textarea**	**Attribute**	**Common**	Defines a **text-area** (a multi-line text input control). A user can write text in the text-area. In a text-area you can write an unlimited number of characters.
93		Attribute	accesskey	
94		Attribute	cols	
95		Attribute	disabled	
96		**Attribute**	**name**	This is of type CDATA and is the form control name or object name when submitted with a form. The name attribute can be used in conjunction with a valid reference to the owning form to reference the corresponding form control (or accessed independently via the id attribute) from within script.
97		Attribute	readonly	
98		Attribute	rows	
99		Attribute	tabindex	
100		***DOM Property***	***form***	*Returns the FORM element containing this control. Returns null if this control is not within the context of a form.*
101		***DOM Property***	***type***	*This read-only property returns the type of this form control. This is the string "textarea".*
102	button	Attribute	Common	
103		Attribute	accesskey	
104		Attribute	disabled	
105		Attribute	name	
106		Attribute	tabindex	
107		Attribute	type	("button"|"submit"|"reset")
108		Attribute	value	
109	fieldset	Attribute	Common	
110	label	Attribute	Common	
111		Attribute	accesskey	
112		Attribute	for	
113	legend	Attribute	Common	
114		Attribute	accesskey	
115	optgroup	Attribute	Common	
116		Attribute	disabled	
117		Attribute	label	
Intrinsic events				
118	a&	Attribute		

119	area&	Attribute		
120	frameset&	Attribute		
121	form&	Attribute	onreset	For use in conjunction with <script> (when the form is reset). This is enabled separately in a custom module without using the Intrinsic events module.
122		Attribute	onsubmit	For use in conjunction with <script> (when the form is submitted). This is enabled separately in a custom module without using the Intrinsic events module.
123	body&	Attribute	onload	For use in conjunction with <script> (when a document loads). This is enabled separately in a custom module without using the Intrinsic events module.
124		Attribute	onunload	For use in conjunction with <script> (when a document unloads). This is enabled separately in a custom module without using the Intrinsic events module.
125	label&	Attribute		
126	input&	Attribute		
127	select&	Attribute		
128	textarea&	Attribute		
129	button&	Attribute		
Metainformation				
130	**meta**	**Attribute**	**I18N**	The **meta** element provides meta-information about your page, such as descriptions and keywords for search engines. Container object for meta in M2. The object and it's attributes are not "properly" implemented in M2. Partial implementation for http-equiv
131		Attribute	content	
132		**Attribute**	**http-equiv**	ContentScriptType, ContentType, Set-cookie and Palette are supported. These values are stored in the httpHeader class that is part of the HtmlDocument DOM object.
133		Attribute	name	
134		Attribute	scheme	
Scripting				
135	noscript	Attribute		
136	**script**	**Attribute**	**charset**	The character encoding used
137		Attribute	defer	
138		**Attribute**	**src**	URL to a file that contains the script (instead of inserting the script into your HTML document, you can refer to a file that contains the script)
139		**Attribute**	**type**	Although there are a number of types (text/ecmascript, text/javascript, text/jscript, text/vbscript, text/vbs, text/xml), I guess we should only be supporting text/jscript?
140		**Attribute**	**xml:space**	Provided by the XmlTextReader. Note that this is NOT set by default.
141		**Attribute**	**-**	Inline scripting support is required.

Link				
142	**link**	**Attribute**	**Common**	The **link** element defines the relationship between two linked documents - commonly used to link to an external style sheet.
143		Attribute	charset	Sets the character encoding
144		Attribute	href	A url to the linked resource
145		Attribute	hreflang	The base language of the linked resource
146		Attribute	media	Which medium the link applies to
147		Attribute	rel	Defines a link relationship from the current document to the linked document; see <a> for values.
148		Attribute	rev	Defines a link relationship from the linked document to the current document; see <a> for values.
149		Attribute	type	The MIME type, like text/jscript or text/css
Base				
150	**base**	**Attribute**	**href**	Defines the default **base** address for relative links
Common				
151	**-**	**Attribute**	**-**	Core + Events + I18N + Style
152	**Core attribute collection**	**Attribute**	**class**	The class of the element.
153		**Attribute**	**id**	String that identifies the object. This must be a unique id for the document or the page will fail to validate.
154		Attribute	title	
		Attribute	**xmlns**	XML namespaces are part of the core supported attributes
155	**I18N attribute collection**	**Attribute**	**xml:lang (and lang)**	Developers are encouraged to use only the xml:lang attributes when specifying the language of an element. but lang is also supported. Note that xml:lang takes precedence over lang (if both are specified) including when xml:lang is inherited from a parent node.
156	Events attribute collection	Attribute	-	These are onclick (Script), ondblclick (Script), onmousedown (Script), onmouseup (Script), onmouseover (Script), onmousemove (Script), onmouseout (Script), onkeypress (Script), onkeydown (Script), onkeyup (Script). These are of limited use in a voice only environment, and hence are not supported.
157	Style attribute collection	Attribute	style	Inline style indicating how to render the element.

10.2.2.1.2 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 setNamedItem, removeNamedItem
- IElement setAttribute, setAttributeNode, removeAttributeNode,

normalize
- IText SplitText

10.2.2.1.3 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.

10.2.2.1.3.1 IE 5,6 event model

Event listener registration:

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

```
<listen id="Listen1" onreco="myhandl()"…>
```

Script method 1: use generic attachEvent method as

```
Listen1.attachEvent("onreco", myhandle);
```

Script method 2: use the event delegate on the object model (OM)

```
Listen1.onreco = myhandle;
```

All above 3 mechanisms share the same event handler:

```
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:

```
<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.

10.2.2.1.3.2 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/. 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:

```
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:

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

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

(Backward compatibility issue in DOM Level 2: it is not clear how one can continue to use HTML "script" tag to register event listener and implement event handler in one stroke.)

10.2.2.1.4 HTML window object

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

Methods:

- attachEvent
- clearInterval
- clearTimeout
- detachEvent
- navigate
- setInterval
- setTimeout

Attributes/Properties:

- length
- 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

10.2.2.1.5 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> and <dtmf> elements. Platforms may then treat such data as applicable to the entire page.

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

```
<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.

10.2.3 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.