Integration Manual

SpeechWorks

OpenSpeech Browser 3.0

Document History

Date	Release Name
August 2003	Third Edition, OpenSpeech Browser 3.0 (Beta Draft)
March 2002	Second Edition, OpenSpeech Browser 1.1
September 2001	First Edition, OpenSpeech Browser 1.0

Published by:

SpeechWorks International, Inc.
695 Atlantic Avenue
Boston, MA 02111

Table of Contents

Chapter 3 Component Integration

Chapter 4 OpenSpeech Browser API

Preface

This book provides information on incorporating OpenSpeech Browser into other applications, such as an interactive voice response platform, and describes the OpenSpeech Browser architecture and components.

Welcome to SpeechWorks

OpenSpeech Browser is a platform integration kit designed to assist integrators in putting the power of VoiceXML in their platforms and networks. OpenSpeech Browser is an open source VoiceXML interpreter, and contains associated internet components for XML parsing, JavaScript execution, and Internet content retrieval. The OpenSpeech Browser and its associated components are provided in binary form for Windows and Linux when you purchase OpenSpeech Browser from SpeechWorks. OpenSpeech Browser is also available in source code, as OpenVXI, from the Carnegie Mellon University.

OpenSpeech Browser can be integrated with other SpeechWorks technologies including the OpenSpeech Recognizer (OSR) and text-to-speech (TTS) engine (Speechify). Other recognition and TTS technologies can also be used.

Roadmap

Recommended reading (available documentation)

Self-starters can learn about SpeechWorks technologies by using the product documentation. The following documents are available for OpenSpeech Browser and Recognizer:

❏ The *OpenSpeech Browser Release Notes* cover specific issues for this release of OpenSpeech Browser, including descriptions of new features, known functional constraints, and migration issues.

❏ The *OpenSpeech Recognizer Developer's Guide* describes grammars: using the built-in grammars and creating your own.

❏ The *OpenSpeech Recognizer Reference Manual* provides details about OSR speech detector and recognizer API functions and configuration parameters.

❏ The *OpenSpeech Recognizer Platform Integration Manual* provides details on integrating OSR with your voice processing platform or VoiceXML browser. It describes the required functionality of the Recognizer, and gives suggestions for managing grammars, caching, and logging.

❏ The *OpenSpeech Recognizer Release Notes* cover specific issues for a release, including descriptions of new features, known functional constraints, and migration issues.

Samples

OpenSpeech Browser includes a set of sample code and grammars examples for using the OpenSpeech Browser functionality. Each sample comes with a Readme.html which describes how to use it.

`installation-directory/samples/`

SpeechWorks Institute

See the SpeechWorks Institute page at http://www.speechworks.com or send email to training@speechworks.com for details about available training courses.

Support services

To receive technical support from SpeechWorks International, Inc.:

- ❑ Visit the Knowledge Base or ask a question at:

  http://www.speechworks.com/training/tech_support.cfm

- ❑ Ask for “technical support” at +1 617 428-4444

How this guide is organized

The chapters of this document cover these topics:

Chapter 1 “Introduction to OpenSpeech Browser” provides information on the overall architecture and provides a guide to reading the chapters.

Chapter 2 “Components and Functionality” describes the components and functionalities of the OpenSpeech Browser.

Chapter 3 “Component Integration” describes the integration of components and hardware.

Chapter 4 “OpenSpeech Browser API” describes the APIs associated with OpenSpeech Browser.

Chapter 5 “Grammars” describes the inline and URL grammars.

CHAPTER 1

Introduction to OpenSpeech Browser

This section provides information on the overall architecture as well as a guide to reading the other chapters.

In This Chapter

Overview

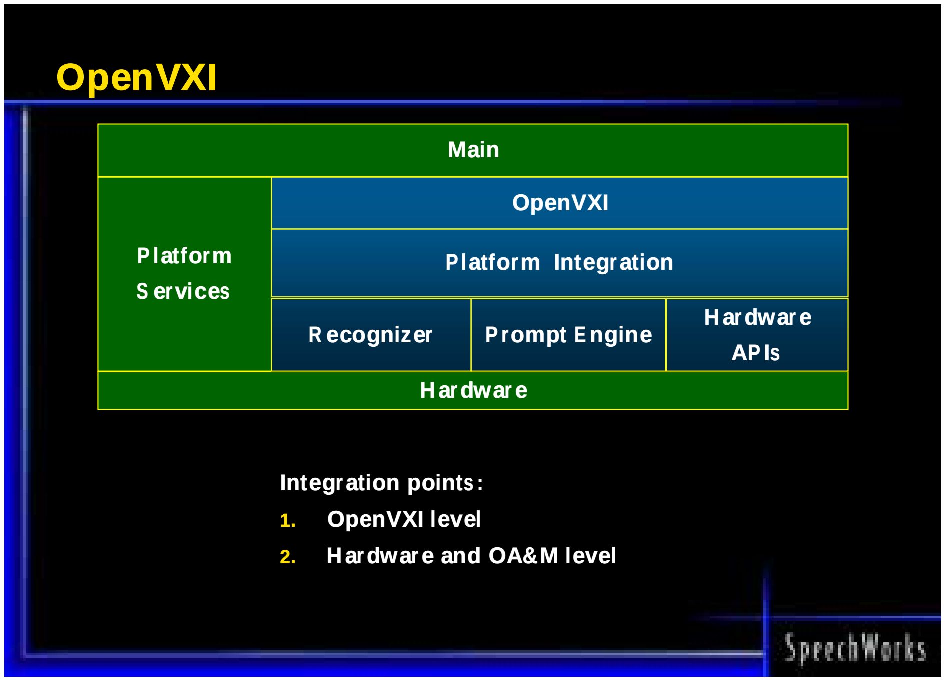

Figure 1-1 OpenSpeech Browser high level integration

OpenSpeech Browser is a platform integration kit designed to help integrators put the power of VoiceXML in their platforms and networks. The core of the OpenSpeech Browser is the OpenVXI open source VoiceXML interpreter and its associated internet components for XML parsing, JavaScript execution, and Internet content retrieval.

OpenSpeech Browser provides only an interpreter and associated Internet components. OpenSpeech Browser relies on three platform APIs for its recognition, prompting, and telephony services. These APIs ("VXIprompt", "VXIrec", and "VXItel") are supported through the OpenSpeech Browser product. Platforms using substantially different models for recognition and prompting services should look at these three OpenSpeech Browser interfaces.

OpenVXI Components

Figure 1-2 shows the components of the OpenVXI.

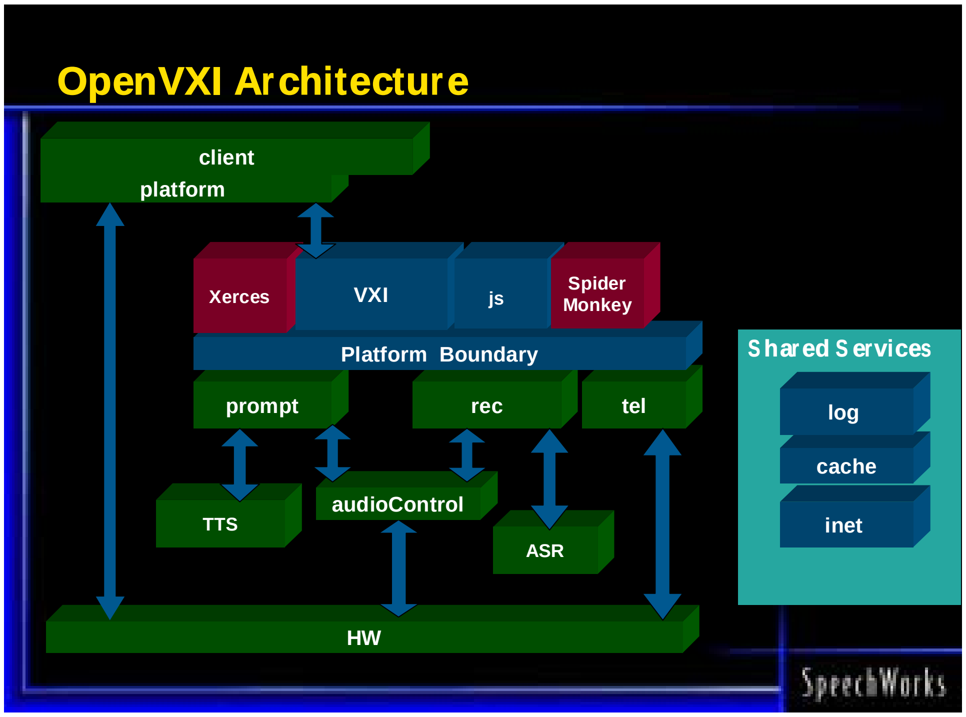

Figure 1-2 Components of OpenVXI

These components organized into three groups.

❏ The blue items are the components provided by SpeechWorks as part of OpenSpeech Browser.

❏ The red items are third party open source software packages that are incorporated into the PIK.

❏ The green items are platform components that should be implemented by the integrator.

The components in Figure 1-2 are:

1. The OpenVXI VoiceXML interpreter (VXI).
2. The Xerces XML parsing engine.
3. A JavaScript (ECMAScript) API and integration to SpiderMonkey (js).

4. A URI and cookie management component for getting data from the Internet (inet).
5. A logging component for writing errors, diagnostics, and events (log).
6. A caching component for writing compiled data back to disk for increased performance (cache).
7. An automatic speech recognizer (ASR), such as the SpeechWorks OpenSpeech Recognizer
8. A text-to-speech (TTS) engine, such as SpeechWorks Speechify.
9. A hardware API, for send commands to and receiving events from a telephony layer.
10. An audio event management and barge-in management component which integrates in the endpointer (audioControl).
11. A telephony component for handling the telephony required by VoiceXML (tel).

Guide to this document

This document is meant as a guide to integrating the OpenSpeech Browser. The complete API documentation is given in Chapter 4.

Chapter 2 discusses the OpenSpeech Browser design and features of each component. Integrators interested only in the interpreter should review the following key components:

Components	Description
VXIinterpreter	The VoiceXML interpreter
VXIrec	The VoiceXML level recognizer API
VXIprompt	The VoiceXML level prompting API
VXItel	The VoiceXML level telephony API

Integrators should review the description and API of the following components:

- “SBcache”
- “VXIplatform”
- “VXIinet”

- "VXIcache"
- "VXIlog"
- "VXIprompt"
- "VXIrec"
- "VXItel"

All integrators should review the information on the interface design in Chapter 2.

CHAPTER 2

Components and Functionality

In This Chapter

Interface design: general principles

The Application Programming Interfaces (APIs), with the exception of the XML parser, are specified as a set of C functions which are grouped together in an interface structure. These interfaces provide a mechanism for filling these interface structures with pointers to functions which implement the required functionality. The functions can be implemented as one or more shared or dynamic libraries. Since the binding of interfaces to the VXI occurs only through the structure pointers, the mechanism by which they are constructed can vary.

The code below shows an example of this structure for the VXItel interface. This interface is found in the header file VXItel.h. The comments in that file are removed here for clarity.

```
typedef struct VXItelInterface {
VXIint32 (*GetVersion)(void);

const VXIchar* (*GetImplementationName)(void);

  VXItelResult (*BeginSession)(
    struct VXItelInterface *pThis,
    VXIMap *args);

  VXItelResult (*EndSession)(
    struct VXItelInterface *pThis,
    VXIMap *args);

  VXItelResult (*GetStatus)(
    struct VXItelInterface *pThis,
    VXItelStatus *status);

  VXItelResult (*Disconnect)(
    struct VXItelInterface *pThis);

  VXItelResult (*TransferBlind)(
    struct VXItelInterface *pThis,
    const VXIMap *properties,
    const VXIchar *transferDestination,
    const VXIMap *data,
    VXIMap **resp);

  VXItelResult (*TransferBridge)(
    struct VXItelInterface *pThis,
    const VXIMap *properties,
    const VXIchar *transferDestination,
    const VXIMap *data,
    VXIMap **resp);
} VXItelInterface;
```

The structure above defines the set of functions required for the VXItel interface. It must be filled by an implementation of the interface such as:

- ❏ The mechanism that created the structure.
- ❏ The pointers that are set to the required functions.
- ❏ Any global initialization required for the component which is defined outside of the interface.

While other interface technologies are suitable for this purpose (C++, Java Interfaces, COM, etc.), a simple C API was chosen for maximum portability across platforms. (Although the API implementations must be implemented as C functions, the platform functionality doesn't. The functions can be used to call local services, components, or remote servers.)

All structured data is passed across interface boundaries as handles. (These may be cast as void* and tested against NULL/0.) The set of types that can be passed as structured data is defined in the Type System section below. The actual implementation of any structured type is invisible across API boundaries.

All of the C API functions, with the exception of the type-system implementation, return a typed result code (VXItelResult in the example above). Each interface defines its own result code type as an enumeration. The result codes are defined so that:

- ❏ Error codes less than zero are program errors either in the VXI or in one of the interfaces.

- ❏ Zero is success.

- ❏ Errors greater than zero are application specific errors which are not a program fault. For example, an error greater than zero is returned when a URI cannot be retrieved.

Type System, strings, and containers

Internally, and in its APIs, OpenSpeech Browser uses an abstract type system to adapt to different architectures and compilers, as well as to support international textual data. These types are defined in the header file vxitypes.h.

OpenSpeech Browser also provides a runtime typed library of abstract data types required by the APIs. Again, these are provided with a C API for maximum portability. These types are defined in vxivalue.h and include both encapsulated basic types and containers. The basic types include:

Type	Description
VXIFloat	An object wrapping around VXIflt32 for incorporation in containers.
VXIInteger	An object wrapping around VXIint32 for incorporation in containers.
VXIPtr	An object wrapping an abstract handle for incorporation in containers.
VXIString	An object implementing immutable strings.

The container types include:

Type	Description
VXIMap	Typedef to VXIMap.
VXIContent	An object that contains binary data with a length and MIME type. Used for audio and other raw information.
VXIMap	Dictionary of key and value pairs.
VXIVector	An object with variable-sized array semantics (like Java Vector).

All container types are polymorphic and the library supports runtime type checking. The base data type to which all the above types can be cast is VXIValue.

VXI

The VXI interface is defined in VXIinterpreter.h. The interpreter's method for executing the content is called Run( ).

Interface function

```
Run( struct
  VXIinterpreterInterface *pThis,
  VXIchar *name,
  VXIMap *sessionArgs,
  VXIValue **result
);
```

Call initiation

When Run( ) is called the interpreter begins processing. The first step in processing is to instantiate the session variable as an object in JavaScript from the information in the *sessionArgs* argument to the Run( ) call. Each key <*key*> in the *sessionArgs* map is added to the JavaScript object and accessed as session.key. For example, the ANI or DNIS information can be retrieved by the client and then added to the *sessionArgs*.

When adding an object as one of the elements of the session, the value of the key in *sessionArgs* should be a *VXIMap*. For example, according to VoiceXML 1.0, ANI and DNIS are to be accessed as session.telephony.ani. To add this to the session variable, the *sessionArgs* must have this structure:

```
{
  telephony {
    ani = <ani value>
    dnis = <dnis value>
  }
}
```

The { }'s indicate the beginning and end of a map; the first element is a key for the VXIMap.

After setting the session variable, the interpreter retrieves the first URL to be run. This URL is supplied as the *name* argument to the *Run* function. It should be a fully qualified URL and should be a web page.

If the first URL is a web page, the interpreter issues an HTTP/1.1 POST for that URL and sends the session variable as an argument. All JavaScript variables sent by the PIK are sent as specified by VoiceXML. Thus objects are sent with the path expanded with "."

NOTE

JavaScript Arrays are simply JavaScript Objects with numbers as the properties. For example, an Array x with two elements is posted to the web server as x.0 and x.1.

The session variable is only sent by default on the first request, subsequent requests are under the control of the VoiceXML pages. The platform integrator can therefore choose to implement the DNIS to URL mapping table either in the client calling Run( ) or behind a web server where the page is derived from the information in the POST.

After retrieving the first VoiceXML page, execution proceeds according to the pages that are retrieved until an exit condition is reached. Upon termination, the interpreter returns the value of an <exit> expression in the variable *result*. If a termination was not caused by an <exit>, the variable is NULL.

Call termination

The interpreter terminates for the following reasons:

- **An <exit> tag is hit.** An <exit> tag causes an immediate normal termination of the run. If a value is set in the exit, the value is returned in the *result* variable.
- **There is no more content to execute.** Execution terminates normally if there is no more content to execute. This happens if a page does not transfer to an additional page. The interpreter returns with a NULL value in *result*.
- **An application error occurs.** If an application error occurs which is not caught as an exception in the calling VoiceXML page, then the default action is to exit. For example if an error.semantic occurs because a page is not a valid VoiceXML, the interpreter returns a NULL value in *result*. If the error is caught by the application programmer, then execution continues as specified in the VoiceXML page.
- **A system error occurs.** If a system error occurs, the interpreter returns with an error code and probably a NULL value in *result*.
- ***[?Jerry: was this removed?]* A recursion or computational limit is reached.** To limit denial of service attacks, the interpreter enforces some constraints on the amount of JavaScript and forms that can be executed without a telephone line and a recognition request. See “Execution constraints” on page 2-13.

NOTE

According to VoiceXML, a <disconnect> does not cause the application to exit. If the event telephone.disconnect is not caught, the default for the interpreter is to exit. If this event is caught, then the application programmer may continue to execute VoiceXML after catching this event.

VoiceXML specifies that the interpreter, in this case *VXIinterpreterInterface*, may continue running after a call has been terminated. VoiceXML requires this behavior so that the client can post information back to the web server or perform additional processing when the call terminates. An active interpreter should not take an additional call in the calling client. If it does, a fatal termination within the interpreter or one of its associated components occurs.

Execution constraints

To allocate resources evenly and minimize security risks, the interpreter enforces the following:

[tbd - 1 - Check ?? below]

❑ Yield on each loop through the Form-Interpretation-Algorithm (FIA) and before loading each page.

This improves scheduling on systems which have thread schedulers that are not exactly fair. These yield points produce the least impact on the caller.

❑ JavaScript is called in one context per thread.

By calling the JavaScript interpreter with one context per thread, the size of the JavaScript interpreters is guaranteed to be the same on each thread.

❑ The interpreter checks through VXItel if a hang-up has occurred before each recognition invocation, on each FIA restart, and before each goto or submit.

[?Removed?] If a hang-up is detected, the interpreter starts a termination counter.

❑ [?Removed?] JavaScript calls are limited to 100,000 loops. (This number is configurable.)

[?Removed?] Limiting the number of loops prevents infinite loop calls in JavaScript.

- [?Removed?] The total number of subdialog calls is limited to 10.

  [?Removed?] To limit infinite subdialog loops, the interpreter does not call subdialogs more than 20 deep.

Default document

The interpreter implements platform defaults through a default VoiceXML document which always loads before any other content and is always active. This document specifies:

- The default language.
- What actions the interpreter should take for events which are not caught by a user VoiceXML page. Most of the default handlers simply play a reprompt or error prompt to the caller using text-to-speech.
- The set of default values for VoiceXML properties.

  If not overridden, the platform defaults compiled into the OSB VoiceXML interpreter are used. To view those defaults, see doc/Defaults.xml in the OpenSpeech Browser installation directory.

  To change the defaults, copy the Defaults.xml into your own working directory, then modify it (keeping the original doc/Defaults.xml). The DTD for this document is a modified copy of the VoiceXML 1.0 [?2.0?] DTD, extended to support language specific defaults, as provided in doc/Defaults.dtd in your OpenSpeech Browser installation directory.

Content validation

After loading a page, the XML is validated using a built-in DTD which is controlled by the VoiceXML version which must be provided at the start of the document in the <vxml> tag. The exact DTD is provided in the document directory located on the CD.

If the document fails validation, an error.semantic is returned in the scope of the VoiceXML page which invoked the page with an error. If the error occurs on the first page, then the default document which is part of the interpreter is invoked.

[?Relevant?] If the VoiceXML version is 1.0, the interpreter permits valid VoiceXML 1.0 with VoiceXML 2.0 working draft extensions.

VoiceXML extensions

Although a number of VoiceXML syntax extensions are supported, none of them are required as the interpreter is designed to run with a pure VoiceXML 1.0 [?2.0?] syntax.

Failing to load a grammar

[tbd - 2 - Tweak this paragraph: we don't ship an OSR integration.]

Grammar load failures are reported on the call to SWIrecGrammarLoad( ), and then again on the call to SWIrecGrammarActivate( ). The following VoiceXML events are generated on errors:

Error	Description
error.grammar	Grammar load, compilation, or activation failed.
error.grammar.inlined	Failed grammar was declared inline (within the doc).
error.grammar.choice	Failed grammar associated with a <choice> element.
error.grammar.option	Failed grammar associated with an <option> element.

The following error events are generated by the implementation:

Event	Description
error..recognition	Failure during recognition attempt.
error..recognition.nogrammars	Recognition attempted with no active grammars.

The error.recognition indicates that something abnormal occurred. In cases where no answer is produced and nothing is heard, nomatch and noinput tags are generated.

Logging

The <log> tag can make the VXI write a user defined log string to the platform logging system. Log is a self-terminating tag with no children. The following attributes are added:

Attribute	Constraint	Description
label	optional	Log label.
expr	optional	ECMAScript to evaluate, result is logged.
cond	optional	If present, this must evaluate to a Boolean result. If the Boolean is true the log is issued.

In addition, <log> takes one or more sub-elements <value>. (see “VoiceXML <log> elements” on page 2-33)

Recognizer property settings

All recognizer properties, provided as <property> tags are passed to the VXIrec interface without interpretation as key-value pairs. The same list of properties is passed to the prompt engine without interpretation.

Internet components

XML parser

The one exception to the principle of C APIs for all components is the integration with the XML parser. Since the interpreter is very tightly coupled to the XML parser, this interface is not as isolated as the defined C APIs. OpenSpeech Browser uses the “Xerces” C++ DOM (Document Object Model) interface published and implemented by Apache (www.apache.org). This implementation is open source and available over the Web.

ECMAScript

The semantics of VoiceXML are deeply coupled to an embedded ECMAScript interpreter since VoiceXML allows the execution of arbitrary ECMAScript scripts. (ECMAScript is an attempt at a standard version of JavaScript, see www.jsworld.com/ecmascript/ for details).

The integration of VXI with ECMAScript is defined by the JSI API. This encapsulates the functionality the VXI requires of JavaScript, allowing the use of any JavaScript engine.

Our default integration uses the “SpiderMonkey” JavaScript codebase from Mozilla (www.mozilla.org). SpiderMonkey provides a C API to compile and evaluate scripts and manipulate objects. The OSB JSI library implements the JSI functions to this API. An implementation to other commercial JavaScript vendors is possible without modification to the VXI.

The file VXIjsi.h specifies the abstract interface for interacting with a JavaScript (ECMAScript) engine. This JSI interface provides functionality for:

- creating JavaScript execution contexts
- manipulating JavaScript scopes
- manipulating variables within those scopes
- evaluating JavaScript expressions/scripts

These functions are called only from within the VoiceXML interpreter. For more information on the interface, refer to the VXIjsi.h header and the open source code.

Internet content retrieval

All content required by any component of OpenSpeech Browser is retrieved through the VXIinet (VXIinet.h) interface. This interface is based on the W3C libwww implementation of HTTP/1.1. OpenSpeech Browser's implementation (SBinet) supports the following major features:

- ❑ HTTP/1.1 persistent connections.
- ❑ Caching of web content including support for content expiration tags and strong and weak content validation.
- ❑ Streaming through content fetches using a file I/O like an API.
- ❑ Cookie support during a call. The cookies can be made persistent across sessions by integrating with two API calls.
- ❑ Higher bandwidth. The libwww implementation is fine for standard HTML browsers, where only one or two sessions are active at a time on any given client, but a VoiceXML interpreter is expected to handle dozens or hundreds of simultaneous sessions. So, SpeechWorks completely rewrote the SBinet component, to remove the serialization and optimize performance. As a result, little remains of the libwww implementation.

Understanding the INET component API requires a solid knowledge of HTTP/1.1: see the IETF specification or information at http://www.w3c.com. A summary of HTTP/1.1 and the relationship to VoiceXML 1.0 is provided below.

VoiceXML and HTTP caching

Two concepts control the caching mechanism for HTTP:

- ❑ A maximum age that a cache entry can obtain.
- ❑ A set of validators for a cache entry which the HTTP server uses to determine if a cache entry is valid.

VoiceXML interacts with the HTTP mechanism using caching properties that can be attached to any piece of data that is retrievable via a URI. The list of properties and their definition for VoiceXML is defined below:

Feature	Description
caching	Either safe to force a query to fetch the most recent copy of the content, or fast to use the cached copy of the content if it has not expired. This attribute's value is used when a value cannot be determined for the maxage attribute and an unexpired copy exists in the cache. If not specified, a value derived from the innermost caching property is used.
fetchtimeout	The interval to wait for the content to be returned before throwing an error.badfetch event. If not specified, a value derived from the innermost fetchtimeout property is used.
fetchhint	Defines when the interpreter context should retrieve content from the server. *prefetch* indicates a file may be downloaded when the page is loaded, whereas *safe* indicates a file that should only be downloaded when actually needed. With either a large file (implying long download times) or a streaming audio source, stream indicates to the interpreter context to begin processing the content as it arrives and not to wait for the full retrieval of the content. If not specified, a value derived from the innermost relevant *fetchhint property is used.

In HTTP/1.1, the cache expiration is set using the maxage or s-maxage HTTP header or indirectly through the EXPIRES response. The result is to determine a freshness lifetime for an object in the cache. If the item in the cache is still fresh, then the HTTP protocol states that the cache entry should be returned. If the cache entry is not fresh, then the client should do a conditional GET on the URI to the web server. The conditional GET is then sent a set of validators for the cache entry. These can be the last modified time, or a set of validators (for example a MD5 encoding of the entry) which where returned by the server.

The server uses these validators to determine if the cache entry is up-to-date and if so, a NOT MODIFIED ON SERVER (304) return code is returned and a new max age parameters to update the freshness lifetime can be sent. In this case, the cache simply returns the cache entry, without fetching the data, and must update the freshness lifetime for the cache entry.

VoiceXML's fast and safe caching properties are tied to HTTP/1.1. Fast is defined to mean use HTTP/1.1 caching mechanism. An implementation must first check if a piece of data in the cache has expired. If it has not expired, the item should be returned to the cache. Otherwise it should conditionally retrieve the item from the URI.

The safe properties tells the VoiceXML interpreter to ignore the HTTP/1.1 caching capability and just do a conditional get on the URI. Note that the same effect can be achieved using fast by making sure that the cache is set to always expire. With

HTML, this is almost always done for pages that need to be refreshed dynamically or that are dynamically generated from scripts. Application implementors can choose which of these two mechanisms to use for controlling the cache.

Most web servers either do not return a maxage or return a maxage of 0 for static content by default. If a maxage is not returned, the client assumes that the maxage is 0. Both of these settings therefore result in the cache always being expired.

It may be possible to adjust this default for particular directories or files using the configuration properties for the web server. Consult your web server documentation for more information.

Cookies in requests

When the browser retrieves a file, the cache reference for the file may have cookie information included with it, which might cause requests to the cache from different browser sessions than the original request to ignore the downloaded file. To fix this problem, ensure all proxies between the server and the client are HTTP 1.1 capable. (For more info see “Controlling Caching” at ftp://ftp.isi.edu/in-notes/rfc2109.txt.) SpeechWorks does not recommend using cookies with grammars and prompts.

Provisioning

In deployment, the following optimizations become a concern:

- Not recompiling information that is already compiled and valid.
- Not fetching information over the network that is already cached and valid.
- Not opening extra sockets to validate information.

Using the HTTP/1.1 Expires header and maxage field parameter allows the customer to accomplish all of these goals.

HTTP caching mechanisms can be used to produce a provisioning service. First introduce a thread in the main program to request pages from the web server that reference all the resources required for an application. Then the interpreter retrieves these resources through the INET component and their HTTP caching properties are set. If the resources are set to not expire in the cache for a provisioned period of time, then callers do not have to wait for the data to be retrieved from the Internet because the data is already resident in the cache.

Cookies

The OpenSpeech Browser stores Cookies in *CookieJars* which are represented as a *VXIVector*. Each element of the vector is a VXIMap that represents properties of a cookie. The properties of the cookie VXIMap match the cookie attribute names as defined in RFC 2965. The only exceptions are as follows:

- INET_COOKIE_NAME: name of the cookie as defined in RFC 2965.
- INET_COOKIE_VALUE: value of the cookie as defined in RFC 2965.
- INET_COOKIE_EXPIRES: expiration time for the cookie as calculated off the MaxAge parameter defined in RFC 2965 when the cookie is accepted.
- RFC 2965 Discard attribute: is never returned in the VXIMap, cookies with this flag set are never returned by *GetCookieJar*.

For definitions of these property names see “OpenSpeech Browser API” on page 4-57.

These functions manage cookies in the VXIinet interface: *SetCookieJar* and *GetCookieJar*.

During a call, if OpenSpeech Browser receives a cookie from the web server, it stores the cookie using the cookie name (supplied by the web server) in an internal VXIVector. While cookie names are usually short and simple, naming conflicts between different web servers and applications are possible if multiple applications and web servers are being contacted by OpenSpeech Browser.

The *GetCookieJar* function allows the platform to retrieve any received cookies then returns a pointer to a new *VXIVector* which contains all the received cookies. The platform then chooses whether to store the cookies persistently and if so, puts them into the internal OpenSpeech Browser *CookieJar* by using the function *SetCookieJar*. This function passes a VXIVector which must contain the set of cookies to be loaded in the format defined above. If cookies are not stored persistently, then this function should still be called with a NULL cookie jar.

Caching

Caching is used by *[?Relevance?]* SWIprompt, the prompting engine, and the recognizer for saving large compiled files. The prompting engine uses it for writing large text-to-speech files back to the disk while the recognizer uses it for writing compiled grammars back to the disk. (This is a general expectation; not all recognizers support a disk cache.)

The *VXIcache* abstract interface for caching is implemented by the SBcache component. The interface is a synchronous interface based on the ANSI/ISO C standard I/O interface. The client of the interface may use this in an asynchronous manner by using non-blocking I/O operations, creating threads, or by invoking this from a separate server process.

Typically, the key name specified by clients is the URL to the source form of the data that is being written back, such as the URL to the grammar text being used as the key name for the compiled grammar. However, the key names specified by clients may be very large in some cases. The most common case is writing back data generated from in-memory source text, such as when writing back the compiled grammar for a VoiceXML document in-line grammar. If the string used for the name is longer than 256 bytes, SBcache uses the MD5 algorithm (specified in RFC 1321 with a full implementation provided) to convert long key names to a 16 byte MD5 value for indexing purposes, using Base64 encoding to convert the binary MD5 value to ASCII text.

The SBcache implementation uses a least-recently-used (LRU) garbage collection algorithm. Whenever a new cache entry is written, if the current cache size plus the size of the new item exceeds the maximum cache size, the cache performs garbage collection in the calling thread. Files are removed until there is sufficient space to write the largest allowed entry into the cache. Items are removed first based on the approximate date of last used, and then by a count on the number of accesses.

There is one cache interface per thread/line.

VXIlog logging facility

The PIK uses an abstract logging facility for all logging. The *VXIlog* interface provides the following types of log streams:

❑ Error stream

Used to report system faults (errors) and possible system faults (warnings) to the system operator for diagnosis and repair. Errors are reported by error numbers that are mapped to text with an associated severity level. This mapping can be maintained in a separate XML document which allows integrators to localize and customize the text without code changes. Also integrators and developers can review and re-define severity levels without code changes. Optional attribute/value pairs are used to convey details about the fault, such the file or URL that is associated with the fault.

❑ Diagnostic stream

Used by developers and support staff to trace and diagnose system behavior. Diagnostic messages are hard-coded text since they are private messages that are not intended for system operator use. Diagnostic messages are all associated with developer defined “tags” for grouping, where each tag may be independently enabled/disabled. Diagnostic logging has little or no performance impact if the tag is disabled, meaning that developers should be generous in their insertion of diagnostic log messages. These are disabled by default, but specific tags can then be enabled for rapid diagnosis with minimal performance impact when issues arise.

❑ Event stream

Summarizes normal caller activity in order to:

- support service bureau billing
- report capacity planning, application and recognition monitoring, and caller behavior trending
- trace individual calls for application tuning

Events are reported by event numbers that are mapped to event names. Optional attribute/value pairs are used to convey details about the event, such as the base application for a new call event.

- ❏ Content stream

  The content stream is a mechanism to log large blocks of data (BLOBs) in a flexible and efficient manner. For example, diagnostic logging of the VoiceXML page being executed, or event logging of the audio input for recognition. To log this information, a request to open a content stream handle is made, then the data is written to that stream handle. Next the client uses a key/value pair returned by the VXIlog implementation to cross-reference the content within an appropriate error, diagnostic, or event log entry. This mechanism gives the VXIlog implementation high flexibility:

  - The implementation may simply write each request to a new file and return a cross-reference key/value pair indicating the file path (key = “URL”, value =)
  - Post the data to a centralized web server
  - Write each request to a database

  The system operator console software may then transparently join the error, diagnostic, and event log streams with the appropriate content data based on the key/value pair.

Across all streams, the log implementation is responsible for automatically supplying the following information in some manner (possibly encoded into a file name, in a data header, or as part of each log message) for end user use:

- ❏ timestamp
- ❏ channel name and/or number
- ❏ host name
- ❏ application name
- ❏ (Error only) error text and severity based on the error number, and the supplied module name
- ❏ (Diagnostic only) tag name based on the tag number, and the supplied subtag name
- ❏ (Event only) event name based on the event number

In addition, for diagnostic logging the log implementation is responsible for defining a mechanism for enabling/disabling messages on an individual tag basis without requiring a recompile for use by consumers of the diagnostic log. It is critical that Diagnostic( ) is highly efficient for cases when the tag is disabled: the lookup for seeing if a tag (an integer) is enabled should be done using a simple array or some other extremely low-overhead mechanism.

It is recommended that log implementations provide a way to enable/disable tags on-the-fly (without needing to restart the software), and log implementations should consider providing a way to enable/disable tabs on a per-channel basis (for enabling tags in tight loops where the performance impact can be large).

Each of the streams has fields that are specified by developers. Several of these fields require following the rules for XML names: it must begin with a letter, underscore, or colon, and is followed by one or more of those plus digits, hyphens, or periods.

However, colons must only be used when indicating XML namespaces, and the “vxi” and “swi” namespaces (such as “swi:SBprompt”) are reserved for use by SpeechWorks International, Inc. The rules are as follows.

❑ Error logging

Module names (moduleName) must follow XML name rules as described above, and must be unique for each implementation of each VXI interface.

Error IDs (errorID) are unsigned integers that start at 0 for each module and increase from there, allocated by each named module as it sees fit. Each VXI interface implementation must provide a document that provides the list of error IDs and the recommended text (for at least one language) and severity. Severities should be constrained to one of three levels:

- “Critical - out of service” (affects multiple callers)
- “Severe - service affecting” (affects a single caller)
- “Warning - not service affecting” (not readily apparent to callers, or at the very least callers do not notice).

Attribute names must follow XML name rules as described above. However, these need not be unique for each implementation as they are interpreted relative to the module name (and frequently not interpreted at all, but merely output). For consumer convenience the data type for each attribute name should never vary, although most log implementations do not enforce this.

❑ Diagnostic logging

Tags (tagID) are unsigned integers that must be globally unique across the entire run-time environment which allows VXIlog implementations to have very low overhead diagnostic log enablement lookups (see above). The recommended mechanism for avoiding conflicts with components produced by other developers is to make the initialization function for your component take an unsigned integer argument that is a tag ID base. Then within your component code, allocate locally unique tag ID numbers starting at zero, but offset them by the base variable prior to calling the VXIlog Diagnostic( ) method. This way integrators of your component can assign a non-conflicting base as needed.

There are no restrictions on subtag use, as they are relative to the tag ID and most log implementations merely output them as a prefix to the diagnostic text.

❑ Event logging

Events (eventID) are unsigned integers that are defined by each developer in coordination with other component developers to avoid overlaps. Globally unique events are required to make it easy for event log consumers to parse the log, so all events should be well-known and well documented.

Attribute names must follow XML name rules as described above. However, these need not be unique for each implementation as they are interpreted relative to the module name (and frequently not interpreted at all, but merely output). For consumer convenience the data type for each attribute name should never vary, although most log implementations do not enforce this.

- Content logging

  Module names (moduleName) must follow XML name rules as described above, and must be unique for each implementation of each VXI interface.

  When a content stream is opened for writing data, a key/value pair is returned by the VXIlog interface. This key/value pair must be used to cross-reference this data in error, diagnostic, and/or event logging messages.

Logging

NOTE

There are two factors in logging that you need to be aware of:

- ❏ Having logging to standard output should only be done for development, not for production.
- ❏ Excessive diagnostic logging degrades performance, so turn it off for production unless absolutely required.

Error logging

OpenSpeech Browser outputs errors based on a module name and error number with associated key/value pairs for additional information such as the URL, etc. Each module allocates its own error numbers, and provides an XML file that maps each error number to appropriate US English text as well as a severity indication (“error – out of service”, “error – service affecting”, or “warning”). This makes it trivial for customers to localize the text and adjust the severity levels for a given platform/ deployment.

Only the module name and error numbers appear in the log file out-of-the-box, not the error text and severities. The real integrations map the error number to text mapping on a remote OA&M workstation rather than within the browser process, thus our out-of-the-box setup mimics it. This also makes logging very low overhead for the browser, making it possible to achieve very high densities even with significant logging configured.

[?Relevance?] By default, the error messages go to the log file configured in the OpenSpeech Browser configuration file where the file format matches the OSR diagnostic/error log. (See the *OpenSpeech Recognizer Platform Integration Manual* for more information about diagnostic and error logs.) Logging is implemented using an error listener that is shipped with OSB in source form: customers may modify it or attach their own error listeners.

[?Relevance?] OSR errors are re-directed out the OSB error log stream through the use of an OSR error listener.

- ❏ Error logging format.

  To get localizable logging, every component defines an error file with the same SB<component>Errors.xml. This is an XML-like file. Each error placed is written as follows:

```
   <ErrorMessages>
<error intf="<moniker>" num="NUMBER" severity="SEVERITY">

   message <xsl:value-of select="@key"/>

   <action> message <xsl:value-of select="@key"> </action>
</error>
</ErrorMessages>
```

  Where NUMBER specifies an error number used in the module in calls to VXIlog::Error. The Number field for error severity is as follows:

Number field	Severity
1	Critical service
2	Service effecting
3	Warning (warning is not an application error since most application errors are logged as severity 2)

  ERROR_MSG is a text message. @Keys can be inserted in the message using the XSL syntax. The message body can use any key that is defined in the error message sent to VXIlog. With this mechanism, the message and the message order can be localized without modifying the code. Furthermore, a simple script can be written to take the Errors.xml file and converted it to a XSL file that produces HTML from the error log.

NOTE

[?Relevance?] If an error appears in the log file generated by the testClient sample application, please consult the relevant log file to determine the source of the error before reporting the issue to SpeechWorks technical support. Also when an issue is reported to technical support, the relevant log file and the SBclient.cfg must be supplied if the problems were generated using testClient.

SBlog

[?Do we still ship SBlog?] The logging interface is implemented by the component SBlog. This component provides a set of listener interfaces with each listener taking the form of a callback. Integrators should write a listener for each type of diagnostic stream and then register that listener for the appropriate type of logging information. When logging information comes into SBlog, it calls all listeners that are registered for that logging information. The listener is then responsible for actually logging the information to a file, database, or other appropriate location. An example implementation of the log listeners is provided in source code as part of the PIK.

SBlog automatically supplies the following information for end listener use across all streams:

- timestamp
- channel name and/or number
- (Error only) error text and severity based on the error number, and the supplied module name
- (Diagnostic only) tag name based on the tag number, and the supplied subtag name

Each of the streams has fields that are specified by developers. Several of these fields require following the rules for XML names: it must begin with a letter, underscore, or colon, and is followed by one or more of those plus digits, hyphens, or periods. However, colons must only be used when indicating XML namespaces, and the “vxi” and “swi” namespaces (such as “swi:SBprompt”) are reserved for use by SpeechWorks International, Inc.

Diagnostic logging

Each component defines its own diagnostic tag IDs that start at the tag ID base plus 0 and increase upwards as required. Each diagnostic tag can be enabled or disabled individually, where the key diagnostic tag IDs are documented. For partners doing their own integration, they can also choose to support enabling/disabling tag IDs on the fly, and enabling/disabling tag IDs for individual channels.

[?OSR-specific; relevant?] OSR diagnostic messages are re-directed out the OpenSpeech Browser diagnostic log stream through the use of an OSR diagnostic message listener. However, OSR diagnostic tag IDs can only be configured through the OSR configuration files, not through OpenSpeech Browser.

[?OSR-specific] By default, the diagnostic messages go to the log file configured in the OpenSpeech Browser configuration file where the format matches the OSR error/diagnostic log. (See the *OpenSpeech Recognizer Platform Integration Manual*.) Diagnostic tags are enabled/disabled at a platform level in the OSB configuration file at start-up time. This is implemented using a diagnostic listener that is shipped with OpenSpeech Browser in source form; customers may modify it or attach their own diagnostic listeners.

Event logging

OpenSpeech Browser generates events based on OpenSpeech Browser-defined event ID numbers (a platform defined range is reserved for integrators). Each event usually has associated key/value pairs for additional information such as the URL, etc.

[?Relevance without OSR integration?] By default, event messages are sent to OSR which writes them to the OSR event log. The OpenSpeech Browser-specific event IDs are mapped to published OSR event names. This is implemented using an event listener that is shipped with OpenSpeech Browser in source form; customers may modify that or attach their own error listeners. That source also includes an implementation that writes events to the OpenSpeech Browser error/diagnostic log rather then sending events through OSR. (For more information on OSR Event Log, see the *OpenSpeech Recognizer Reference Manual*.)

[?Reference to SpeechGenie?] The table below lists the events that OpenSpeech Browser logs. Unless otherwise indicated, this matches the SpeechGenie platform as defined in the SWI and VoiceGenie OA&M and Event Logging Specifications. The OpenSpeech DialogModules and/or the application provide the application call flow logging required for using OpenSpeech Insight.

In addition, OpenSpeech Browser does not bundle any tool for converting event logs to a form compatible for SpeechWorks OpenSpeech Insight.

OSB event ID	OSR event name	Description	Keys
0	SWIclst	Call Start	ANII (ANI), DNIS (DNIS), VURL (initial VoiceXML document URL)
1	SWIclnd	Call End	N/A
2	SWIbrsr with BREV=2oras app specified	Log Element (<log>)	As specified by the VoiceXML application
3	SWIendp	End pointer log (speech detected)	BRGN (whether barged in), BTIM (speech start time), both as defined by the OpenSpeech Recognizer.
4	SWbrsr with BREV=4	VXML document fetch	VURL (VoiceXML document URL)

[?OSR-specific info] OpenSpeech Browser looks for the following OSR-defined properties within VoiceXML applications to support proper OSR logging. When present, these OSR parameters are set prior to the next SWIrecGrammarLoad( ), SWIrecGrammarActivate( ), or SWIrecRecognizerStart( ) call.

NOTE

[?OSR-specific info] OpenSpeech Browser passes all “swirec_*” properties through to OSR, so new OSR logging controls in future OSR releases become available under OpenSpeech Browser immediately.

Property	Action
swirec_application_name	Calls SWIrecRecognizerSetSessionName( ) with this value. This allows OSR to log the application name for each recognition as required by LEARN for platforms running multiple applications.
swirec_suppress_event_logging	Passes this parameter through to OSR to disable event logging.
swirec_suppress_measurement_logging	Passes this parameter through to OSR to disable measurement logging.
swirec_suppress_waveform_logging	Passes this parameter through to OSR to disable waveform logging.

[?OSR-specific info] OSB also implements floating waveform capture. The reference application manager provides the full waveform capture implementation in source form with the OpenSpeech Browser distribution, allowing platform integrators to

add additional functionality such as waveform capture only for particular applications, particular languages, portions of applications, or particular OSDMs. The controls are implemented in the OpenSpeech Browser configuration file as follows:

Configuration setting	Description
waveformCaptureLineLimit	Maximum number of lines to enable waveform capture on. OpenSpeech Browser enables waveform logging for the entire duration of calls, ensuring that no more then this number of lines have waveform capture enabled. (Event logging and measurement logging is on or off for all lines as configured through OSR at the platform level.) Note that OpenSpeech Browser ensures that waveform logging cannot be turned on by an application by setting swirec_suppress_waveform_logging to FALSE: that parameter is suppressed on lines where waveform capture is disabled at the platform level.

VoiceXML <log> elements

VoiceXML <log> elements are implemented in OpenSpeech Browser. The <log> element may be embedded within the VoiceXML application with the following information:

Data	Description
label attribute	Log label
expr attribute	ECMAScript to evaluate, result is logged
<log> element content	Data to log, may include variables that are resolved

OpenSpeech Browser generates separate event and diagnostic log lines for each of label, expr, and the <log> element. The rules for doing so are provided below, where these rules are implemented in the default event log listener provided with OpenSpeech Browser in source form for easy customization.

NOTE

Both diagnostic messages and events are generated for each <log> element. Since diagnostic tags can be disabled from the OpenSpeech Browser configuration file *[?Does OpenSpeech Browser 3.0 have a config file?]*, don't disable the diagnostic logging just to avoid a performance penalty. The event logging could also be disabled, which requires a change to the default event listener code supplied with OpenSpeech Browser.

Diagnostic logging

1. For the "label" attribute (if present), a diagnostic log entry is written with a tag ID of 8001 (VXI base + 1) with text of the form "label=<text>" where <text> is replaced by the application specified text.

2. For the "expr" attribute (if present), the expression is evaluated and a diagnostic log entry is written with a tag ID of 8001 (VXI base + 1) with text of the form "expr=<text>" where <text> is replaced by the evaluation results.

3. For the <log> element content (if present), any variables are evaluated and a diagnostic log entry is written with a tag ID of 8001 (VXI base + 1) with text of the form "content=<text>" where <text> is replaced by the application specified text.

Event logging

1. For each of the “label”, “expr”, and the <log> element content, if the final text looks like a valid OSR event log entry (starting with “EVNT=” followed by an event name followed by zero or more key/value pairs):

    a. OpenSpeech Browser does minimal processing to resolve white space issues caused by the VoiceXML interpretation and XML parser.

    b. OpenSpeech Browser sends the event text to OSR for logging. *[?OSR-specific]*

2. Otherwise for each of the “label”, “expr”, and the <log> element content, if the final text is raw text:

    a. OpenSpeech Browser does processing to resolve white space issues caused by the VoiceXML interpretation and XML parser.

    b. OpenSpeech Browser escapes occurrences of '|' with a second '|' as required by the OSR Event Log file format. *[?OSR-specific]*

    c. OpenSpeech Browser replaces occurrences of '=' with '->' to escape them. The OSR Event Log file format does not permit using '=' within key names or value names.*[?OSR-specific]*

    d. OpenSpeech Browser writes a SWIbrsr event (“SpeechWorks browser”) with a key/value pair of BREV=2 (“browser event #2”) and a second key/value pair of either “label=<text>”, “expr=<text>”, or “content=<text>” as appropriate where <text> is replaced by the final text.

Hardware interfaces

OpenSpeech Browser uses three hardware interfaces to communicate with the hardware platform.

- Audio Player – hardware audio player interface
- Audio Source – hardware audio source interface
- Session Control – hardware session control and DTMF interface

For OpenSpeech Browser to run on the hardware platform, these interfaces must be implemented by the platform integrator.

The session control interface is the primary place where call control extensions or additional call handling capabilities are added. The implementation of these interfaces must be modified to support new hardware platforms.

Platform resource interfaces

Recognition interface

The recognition interface, called VXIrec, provides the recognition resource for the VoiceXML interpreter. This can be difficult to implement, because it must perform some complex grammar management to fully implement the VoiceXML specification. Integrators of the OpenSpeech Browser do not need to work with this API, but this information is provided for completeness.

[tbd - 3 - [?Last statement makes no sense]

There is one recognition interface per thread/line.

Prompt interface

SBprompt interface is an implementation of the VXIprompt abstract interface for prompting functionality. VoiceXML allows both pre-recorded and TTS synthesized prompt segments which can consist of a mixed sequence of these segments. Pre-recorded audio segments are referenced as URIs, while TTS segments consist of text with optional markup. The distinction between prompts and segments is that several VoiceXML properties (counts, barge-in, etc.) apply at the prompt level, not the individual segment level.

Prompts are constructed as a series of audio segments, where segments may be:

- in-memory audio samples
- paths to on-disk audio files
- URLs to remote audio
- text for playback via a Text-to-Speech engine
- text for playback using concatenative audio routines (123 played as “1.ulaw 2.ulaw 3.ulaw”).

The Prompt interface handles prefetching, caching, and streaming audio as required to provide good response times and low CPU and network overhead.

Telephony interface

VXItel provides the telephony functions for the VXI. The transfer type is split into the bridge and blind transfers. These platform functions are very platform and location dependant.

Object interface

The Object Interface is basic and, outside of initializations, consists of only one call objectExecute( ). The VXI collects all parameters and arguments from attributes into respective VXIObject's and calls objectExecute( ) which locates and invokes the required platform objects. Upon completion, it returns a VXIMap containing its results which is imported by VXI into the JavaScript environment.

Execution models

For all, if a resource is getting put in a separate process from resources it depends on, then the integrator needs to build a proxy that convert functions calls to IPC request/ response pairs.

SBvalue, SBjsi, SBprompt, SBrec, and SBtel are all passive and don't care what the execution model is and each resource instance is independent of all the others. SBhw uses a single thread to handle all the hardware IO and does require threading.

Multiple process implementations require some additional component replacements. All of the components depend upon a base threading library in *VXItrd.h.* The implementation dll SBtrd.dll is bound to each component requiring threads or mutexes. Replace this for any form of multi-process implementation in order to replace the critical sections and mutex with kernel mutexes and inter-process semaphores.

[tbd - 4 - Do we still ship this library?]

SBinet functions in all cases, but there is a separate cache for each process that hosts the SBinet implementation. For multi-process environments, a single cache requires an integrator provided proxy to put all SBinet implementations into a single process, or full integrator replacement. The same is true for SBcache.

CHAPTER 3

Component Integration

In This Chapter

Getting started integration

A typical integration involves at least the following:

- ❏ Integrating the components of the OpenSpeech Browser into the platform main and any platform services.
- ❏ Integrating to a recognizer, a prompting engine, and a telephony layer by implementing to the VXIrec, VXIprompt, and VXItel interfaces.
- ❏ Integrate the required OpenSpeech Browser components into the platform main and any platform services.
- ❏ Integrate the logging interface to get diagnostic, errors, and events delivered into the platform.

In addition, some integrators might want to replace or extend the caching or Internet I/O components.

Viewing and building reference source on Windows

[tbd - 5 - [?Correct this section for actual source code and development environment provided]

OpenSpeech Browser provides source code to get integrators started with the hardware, logging, and client integration steps. A *src* subdirectory is installed in the main SpeechBrowser directory. By default this is found in C:/Program Files/SpeechWorks/OpenSpeech_Browser_PIK/src. This directory contains a subdirectory for the available sample hardware integrations and the for the client. The source in the client directory includes a sample implementation of a set of logging listeners.

Each of these directories contains a Microsoft Visual Studio Project file for Visual C++ version 6.0. Open and explore these projects to examine the files that are involved in each integration and see how they relate to the rest of the code base.

When the Visual Studio Projects are executed, they build DLLs or EXEs and put them in subdirectories below the project file. The files are placed in the sub-directory *Debug* for debug builds, and *Release* for release builds. To run these DLLs or the executable, modify your PATH environment variable so that the newly built files appear in the computer's path prior to the OpenSpeech Browser directory. Changing the path is not the only mechanism, but is recommended to prevent overwriting the original files provided by OpenSpeech Browser.

[?Fix this example; no HW anymore] For example, if the default installation directory is used, the hardware project appears in C:/Program Files/SpeechWorks/OpenVXI/src/hw/dialogic. The built debug version of the hardware DLL is in C:/Program Files/SpeechWorks/OpenVXI/src/hw/dialogic/Debug/SBhw.dll. The directory C:/Program Files/SpeechWorks/OpenVXI/src/hw/dialogic/Debug is then put in the front of the PATH. When the testClient program is run, this new custom integration DLL gets used instead of the one that came as part of the package.

Viewing and building reference source on UNIX

[tbd - 6 - [?Correct this section for actual source code and development environment provided]

OpenSpeech Browser provides source code to get integrators started with the hardware, logging, and client integration steps. A *src* subdirectory is installed in the main OpenSpeech Browser directory. By default this is found in /usr/local/SpeechWorks/OpenVXI/src. This directory contains a subdirectory for the available sample hardware integrations and the for the client. The source in the client directory includes a sample implementation of a set of logging listeners.

Each of these directories contains a GNU make makefile for GNU gcc 3.0. Open and explore these directories to examine the files that are involved in each integration and illustrates how they relate to the rest of the code base.

When the makefiles are executed, they build libraries or executables and put them in subdirectories below the makefile. The files are placed in the sub-directory *Debug* for debug builds, and *Release* for release builds. To run these libraries or the executable, modify your PATH environment variable so that the newly built files appear in the computer's path prior to the OpenSpeech Browser directory. Changing the path is not the only mechanism, but is recommended to prevent overwriting the original files provided in the OpenSpeech Browser.

For example, if the default installation directory is used, the hardware source appears in /usr/local/SpeechWorks/OpenVXI/src/hw/dialogic. The built debug version of the hardware library is in /usr/local/SpeechWorks/OpenVXI/src/hw/dialogic/Debug/SBhw.a. The directory */usr/local/SpeechWorks/OpenVXI/src/hw/dialogic/Debug* is then put in the front of the PATH. When the testClient program is run, this new custom integration library gets used instead of the one that came as part of the package.

Integrating the components into a new client

[tbd - 7 - [?Fix this section for 3.0 install]

OpenSpeech Browser is designed to be integrated into a platform. The source code to the test program *testClient* is located in the src/client subdirectory of your installation. On Windows, the Microsoft Visual C++ project workspace file *testClient.dsw* contains a project file for both the platform DLL *SBclient.dll* and for the test executable *testClient.exe*. On UNIX, the makefile builds both the platform library *SBclient.a* and the test executable *testClient*.

The platform DLL/library is designed to facilitate integrations by showing how all the components of the PIK should be initialized, shutdown, and managed for each thread. The file *SBclient.cpp* contains all the code for:

- Initializing all the components of the PIK.
- Shutting down all the components of the PIK.
- Creating and destroying the components for the different threads.
- Setting the properties for the logging and other components from information in the configuration file.
- Initializing the *VXIsessionControl* and waiting for a call.
- Taking a call and then running the interpreter.
- Initializing log listeners.

Review this code with the API documentation to determine how to initialize all the components.

The file SBlogListners.cpp contains a sample implementation of a *SBlog* log listener. A listener is provided for each of the logging types: Events, Diagnostics, and Errors. Events are written to the OpenSpeech Recognizer event log using the OpenSpeech Recognizer function SWIrecLogEvent( ), which is done through an abstraction interface and the actual call in the code i:

```
rec_->LogEvent
```

SpeechWorks recommends that any logging integration maintain the sending of events into the SpeechWorks' event log.

The information in the event log is used by OpenSpeech Insight (OSI) and the OpenSpeech Recognizer's LEARN process. SpeechWorks provides a set of OpenSpeech DialogModules (OSDM) to simplify application development and tuning. The OSDM makes use of the <log> tag supported by the OSB. This <log> tag log information is written to the Event logging stream and becomes part of the SpeechWorks event log. OSI can then be used to get statistics about call completion and dialog completion rates from the event data.

Hardware integration

OpenSpeech Browser uses three hardware interfaces to communicate with the hardware. The platform must support these three interfaces for OpenSpeech Browser to function.

- ❏ Audio Player – hardware audio player interface
- ❏ Audio Source – hardware audio source interface
- ❏ Session Control – hardware session control and DTMF interface

The session control interface is the primary place where call control extensions or additional call handling capabilities are added. The implementation of these interfaces must be modified to support new hardware platforms.

When writing the Audio Control, Audio Source, and Audio Player implementations, think about what threading model to use. Do not deliver events on the kernel thread used to provide your hardware event notifications, as OpenSpeech Browser frequently does significant processing on the thread between the point when the OpenSpeech Browser event handler (callback) is invoked and that handler returns.

For example, when AP_EVENT_COMPLETED events are delivered, OpenSpeech Browser does state machine processing on that thread, and may often initiate additional audio fetches and audio plays prior to returning. TriggerRecord( ), endpointing, recognition, and TriggerStop( ) are called for AS_EVENT_RECEIVED events before the event handler exits.

It is important to use a non-kernel thread to avoid blocking hardware drivers and the kernel, and that the integration be re-entrant, including avoiding deadlocks. One practical solution to this is to use separate threads to deliver events which are queued for delivery on the kernel thread and the other “event” threads reading from the queue and delivering those events.

Audio player

The AudioPlayer interface is used by OpenSpeech Browser to play audio out to the hardware. The interface is designed as an asynchronous event based interface to maximize performance. Three states to the interface are:

- ❑ Idle

  In the idle state, the hardware interface is not performing any work. This is the starting state and the state to which the system should transition on any errors.

- ❑ Playing

  In this state, the interface is actively delivering audio from files or buffers to the hardware. When the interface is in the playing state it must not accept additional buffers or files to play.

- ❑ Streaming

  In this state, the interface is also actively delivering audio to the hardware. However this state only excepts buffers to be played, and it must accept additional stream request while playing.

Transitions between the states are driven by messages sent as function calls to the API and by internal events. On each state transition, an event is returned to all registered event listeners so that external components can track the state of the interface. This section provides details on all of the messages and events that drive the state machine. The function and event names in the figure are abbreviated for clarity.

Figure 3-1 is an abstract state machine for delivering audio to the hardware. The hardware must implement this interface for OpenSpeech Browser to deliver prompts.

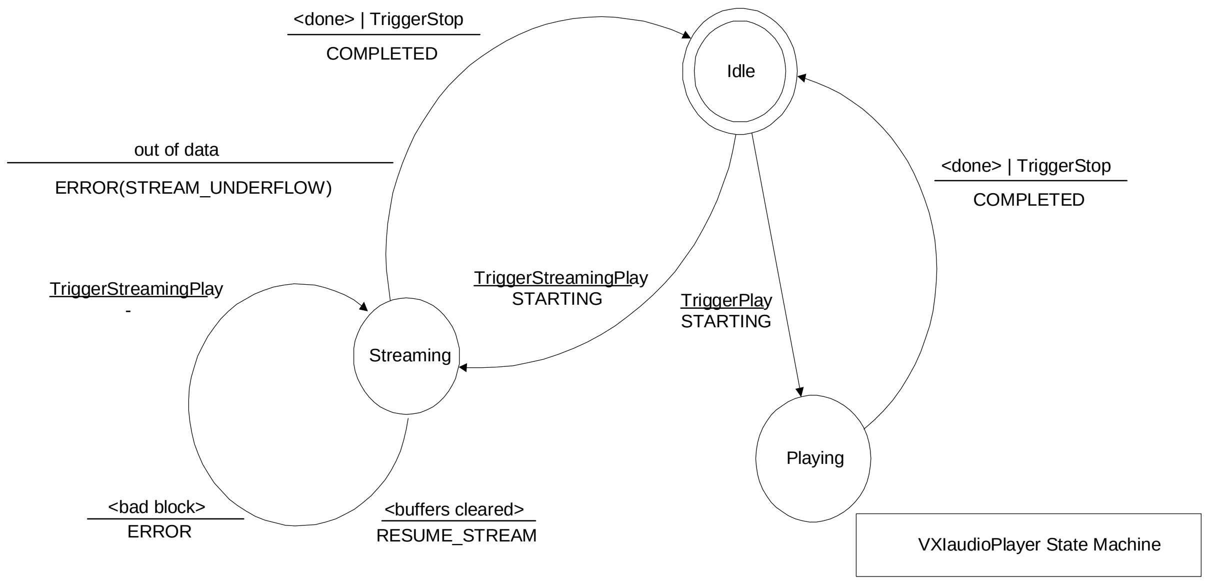

Figure 3-1 VXIaudioPlayer State Machine

One of the key requirements for the audio player is that the time to respond to a TriggerStop( ) be less then 200 msec. Otherwise callers may think the system did not hear them, which could cause the caller to stutter, which causes recognition accuracy issues.

For “normal” (non-streaming) audio player plays performed via TriggerPlay( ), the VXIaudioPlayer interface makes integration easier by removing any requirement for queuing audio yourself. Instead, the OSB passes a linked list of audio for you to play, and delivers more audio after you report that the play is complete through an AP_ PLAY_COMPLETED or AP_PLAY_ERROR event.

The nodes in the linked list varies in the MIME content type and also in whether the data is in-memory or in a file. OpenSpeech Browser delivers in-memory audio when it plays back audio it recorded from the caller, or when the audio was fetched from a remote web server. It delivers a path to a file when the audio referenced by the application is a file on the local system, as many telephony libraries are optimized for file-based playback (performing caching, optimized buffering, etc.). File paths are always absolute operating system dependant file paths for direct use in fopen( ) or equivalent calls. When doing playback for TriggerPlay( ), there can be gaps between the chunks of audio played from a play list, although it is desirable from a UI perspective to minimize them.

The MIME content type for each block of audio varies depending on the application. OpenSpeech Browser does not do MIME content type conversion or filtering: it merely passes through the path or in-memory data with the MIME content type reported by the remote web server or through extension mapping rules defined in [?this file no longer exists] SBclient.cfg (or as passed to SBinetInit( )). This permits your implementation to define and extend the supported audio types completely independent of OpenSpeech Browser.

However, you should support the following MIME content types:

audio/basic	Headerless, 8 kHz 8-bit mono mu-law
audio/x-alaw-basic	Headerless, 8 kHz 8-bit mono a-law
audio/wav	WAV, 8kHz 8-bit mono, mu-law or a-law

Document the supported MIME content types for the application developers, so they can configure their application web servers to return the proper MIME content type strings (for most web servers configured as extension mapping rules).

One key requirement of the Audio Player is the ability to support streaming audio plays (TriggerStreamingPlay( ). While not necessary to implement (you may return VXIap_RESULT_UNSUPPORTED), if you do, text-to-speech playback always fails. When building a streaming audio implementation, consider the following.

❑ First, your implementation should pre-buffer some amount of audio in order to ensure a real-time audio stream to the caller. While TTS engines such as Speechify typically return the TTS stream faster then real-time, it is possible that under high loads the TTS server may slow down, or OpenVX may have sporadic delays in the delivery of the audio stream. By pre-buffering some amount of audio before actually beginning the play to the caller, such as one-half to one second of audio, you create a backlog that helps avoid underruns (audible gaps in the stream). Many hardware libraries have dedicated mechanisms for streaming audio that include doing such pre-buffering.

❑ Next, the AP_EVENT_STARTING should only be delivered once the actual audio becomes audible to the caller, or as close as possible. This permits OpenSpeech Browser to know when the caller is hearing audio in order to provide extensions in the future that require this knowledge.

Once the streaming play is initiated, your implementation is also responsible for managing a list for storing the streaming audio. Each call to TriggerStreamingPlay( ) delivers one chunk of audio, where the VXIaudioPlayer interface guarantees that all the audio in each stream is in-memory audio of a single MIME content type, and that non-streaming plays is not performed in the middle of a stream. As each packet is delivered, you can attach that packet to the prior packet by using the “next” and/or “previous” pointers as you see fit (you may also choose to use your own external queuing mechanism).

While proceeding with the streaming play, call the Destroy( ) method of each packet to release that memory back to OpenSpeech Browser (you can leave the “next” and “previous” pointers as-is, OpenSpeech Browser ignores those and only releases that single node). If possible, free a group of packets on a scheduled basis, but if this is not possible, hold on to all the packets until the play terminates, then free them all at the end (although this increases memory consumption).

For flow control purposes, you can return VXIap_RESULT_STREAM_ OVERFLOW from TriggerStreamingPlay( ) to refuse an audio packet, telling OpenSpeech Browser to hold on to that audio packet and pause delivery of the streaming audio stream until your implementation delivers an AP_EVENT_ RESUME_STREAM event (or the audio stream completes and you deliver an AP_ EVENT_COMPLETED event). It is recommended that you use a low watermark and high watermark approach, where the high watermark indicates the buffer size at which you pause the audio stream (recommended 30 to 60 seconds), and the low watermark indicates the buffer size at which you resume the audio stream (recommended 10 to 30 seconds).

Keeping the high watermark low reduces memory use but increases the risk of an underflow (audible gap) if there are sporadic delays in the audio stream delivery. Keeping the low watermark significantly below the high watermark, increases CPU efficiency between OpenSpeech Browser by allowing it to send a group of audio, then “rest” for a while, then send another group of audio, then “rest” more, rather then consuming CPU space by continually alternating the OpenSpeech Browser state machine between a stream pause and stream resume state.

If you detect an underflow (gap) during an audio play, notify OpenSpeech Browser so it logs a notification, and lets it take appropriate actions to avoid future underflows. You do this through one of two events:

- AP_EVENT_WARNING with a status code of VXIap_RESULT_STREAM_ UNDERFLOW to report the warning but allow the streaming play operation to be active.

- AP_EVENT_ERROR with a status code of VXIap_RESULT_STREAM_ UNDERFLOW to report that the underflow caused the streaming operation to stop and thus the Audio Player is returning to an idle state.

In the case of AP_EVENT_ERROR, OpenSpeech Browser starts a new streaming audio operation that picks up where it left off. The only difference is with AP_ EVENT_ERROR, you can handle a telephony library that automatically aborts the stream, and also have the opportunity to pre-buffer audio again to avoid future underflows at the cost of a large audible gap for the caller.

Audio source

There are two ways to use detect speech.

- ❏ The first is using an external first-pass speech detector, which reduces CPU overhead by using hardware to look for speech, and not delivering any audio back to OpenSpeech Browser for recognition until that speech is detected.

  For example, most telephony board vendors support a speech detector running on DSPs on the board. When the Audio Source TriggerRecord( ) operation is started, the hardware-based speech detector is enabled and the SBhw implementation waits for the hardware to deliver a speech detected event.

[tbd - 8 - [?This probably needs work]

  When the speech is detected, without cutting off any playing prompt, the SBhw implementation then delivers audio samples back to OpenSpeech Browser for processing. OpenSpeech Browser uses the endpointer to do a second state of speech detection and endpointing (to avoid false barge-in). Once the endpointer verifies speech and properly trims leading silence, OpenSpeech Browser then cuts off the prompt by calling the Audio Player TriggerStop( ) method.

- ❏ The second is to use hardware without on-board speech detection. Instead it immediately returns audio once TriggerRecord( ) is called. This option, with the additional real-time processing across all the channels, takes a significant CPU toll.

  When a first-pass speech detector is being used, refer to your recognizer vendor's manuals for a detailed discussion of how it works. In particular, it is important with most recognizers to buffer audio so at least 200 msec of audio prior to speech being detected is delivered. This is because most speech detectors trigger on vowels; this pre-speech buffering allows the recognizer to ensure leading consonants are not trimmed.

The VXIaudioSource interface is used to receive audio from the hardware. Audio is received in blocks through events which are delivered by the hardware. The data returned by the hardware is sent to the end-pointer and then if speech is detected, it is sent to the recognizer. The audio source assumes that some computation is done within the event callback that it uses to deliver the audio. Thus a kernel level thread cannot be used for the audio source event thread.

OpenSpeech Browser assumes that the state machine shown in Figure 3-2 is implemented by the audio source interface. Transitions in this state machine are driven by function calls from OpenSpeech Browser and internal events. When the system starts up it should be in the Idle state. After registering for events, OpenSpeech Browser calls the function TriggerRecord( ) to start the receipt of audio which transitions the audio source into a Recording state. From this state, whenever audio is received it calls the registered event callback to deliver the audio. The event callback can return either *SUCCESS* to indicate that this was used and more should be delivered, or *PAUSE_AUDIO*. If *PAUSE_AUDIO* is returned, it means the audio was not consumed and the audio source should stop the delivery of audio by transitioning to a Paused state.

When OpenSpeech Browser is ready to take more audio in the Paused state, it calls the function TriggerResume( ). The audio source is returned to the Idle state either by an error or a call to TriggerStop( ). Whenever the audio source transitions to the Idle state it must throw the STOPPED event through the event callback.

The OpenSpeech Browser API provided details on the different function calls and events.

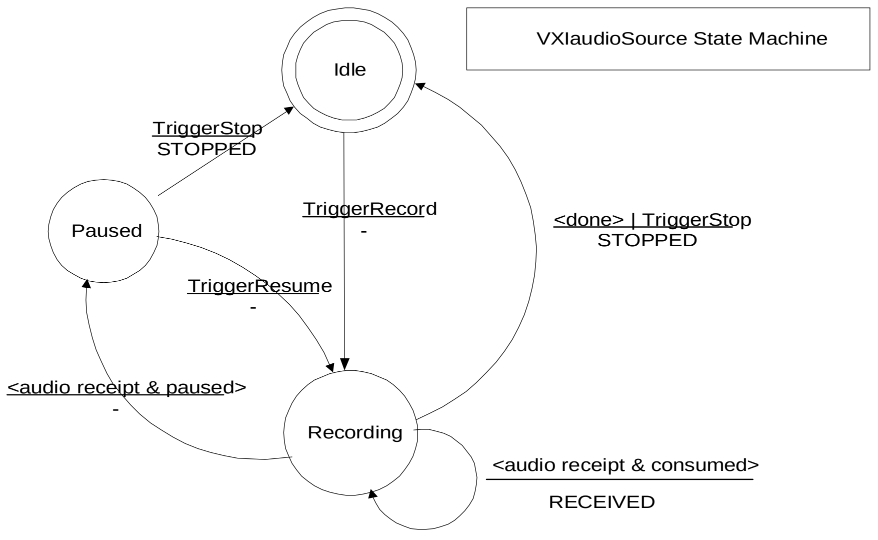

Figure 3-2 VXIaudioSource State Machine

Session control

The VXIsessionControl interface (Figure 3-3) is the call control and DTMF handling interface for OpenSpeech Browser. OpenSpeech Browser uses this interface to receive DTMFs in the audio processing component, to handle transfer and hang-up from the VXItel component, and to receive the initial call in the platform component. This is the primary location where integrations should put additional call control functionality.

For example, if out bound calling is to be supported, a function can be added here to support outbound dialing. These functions may also be performed outside of this interface.

Figure 3-3 shows the basic state diagram and function call and event driven transitions. When the system first initializes, the interface and the hardware itself must be placed in an OUT_OF_SERVICE state. Disable any channels in this state and treat them as “busy” by a switch. Placing the session control in this state should also disable the audio source and audio player interfaces.

In the state machine for the session control (Figure 3-3), an error always goes to the OUT_OF_SERVICE state which puts the phone line off-hook if possible.

Once all of the lines are up, the platform client *[?Not sure the word "client" is relevant or meaningful here; maybe "telephony layer" would be clearer? Or just "platform code"?]* code calls WaitForCall( ) on each line, causing a transition to the IDLE state in the session control. The function call is non-blocking.

When a call is received, the session control delivers an INVITE event to its registered event listeners. There are typically three event listeners per-channel in OpenSpeech Browser.

When OpenSpeech Browser is ready to take the call, the platform client code calls Accept which transitions the session control into the IN_CALL state. This state can deliver DTMFs to the event listeners when they are received.

After a call is picked up, the client calls GetNthPropertyPair( ) to load all the line properties into a VXIMap. This is then passed to the VXI in the Run call so that the VXI can put this information into the VoiceXML session state.

If the function call Terminate( ) is called or the call originator terminates the call, then the session control must return to the OUT_OF_SERVICE state. Since OpenSpeech Browser may continue voiceXML processing after call termination, this is essential to prevent a call race condition.

The session control transitions from IN_CALL to Transfer if the Transfer( ) function is called. The Transfer( ) function is blocking.

There are two versions of the *Transfer* function: blind and bridged.

In a blind transfer, the session control performs the transfer and if this results in the call originator call portion leaving OpenSpeech Browser, the session control also delivers an END event and transitions to OUT_OF_SERVICE.

A bridge transfer transitions to the Transfer state and then stays there for the duration of the bridge transfer. If the called party terminates or the duration of the transfer is exceeded, the session control should return to IN_CALL. This transition must throw the TRANSFER_COMPLETE event and also cause the return from the Transfer function call. If the calling party terminates the call from within a bridge transfer, then a transition to OUT_OF_SERVICE must occur with the corresponding delivery of the END event.

Transfer functionality generally requires integration work that depends upon the telephony environment for the platform. OpenSpeech Browser delivers the transfer destination as a URI as provided directly within the VoiceXML page. The interpretation of this URI is left to the implementation of the session control.

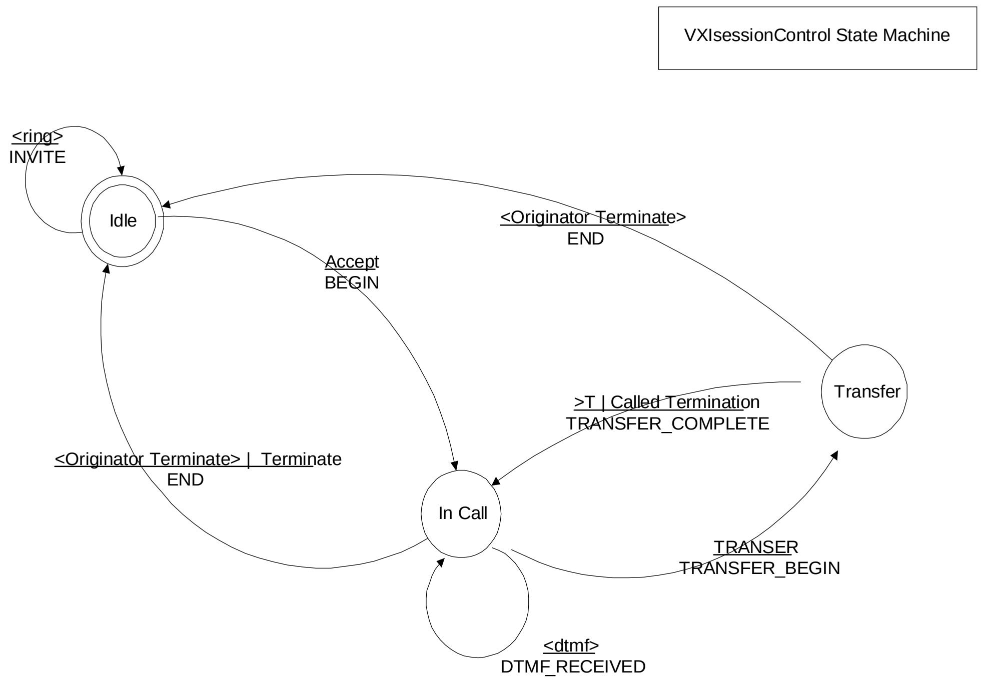

Figure 3-3 VXIsessionControl State Machine

When the session control receives a hang-up from the hardware an implementation may also terminate the audio player and audio source. This is not required, but is allowed in the integration. In this case, the audio source should return STOPPED and audio player should return COMPLETED.

Implementing custom fetching and caching

This section is a high-level overview of the mechanisms for replacing the OpenSpeech Browser implementations of URI fetching and caching. VXIinet and VXIcache are two separate interfaces through which OpenSpeech Browser fetches grammars and caches compiled grammars. OpenSpeech Browser comes prepackaged with its implementations of these interfaces, but users may replace one or both interfaces with custom implementations. VXIinet and VXIcache may be replaced for the following reasons:

❑ To implement a different caching algorithm or to provision grammars based on specialized knowledge of application behavior.

❑ To implement a different caching store, for example, a networked grammar cache which can be shared across multiple recognizer machines.

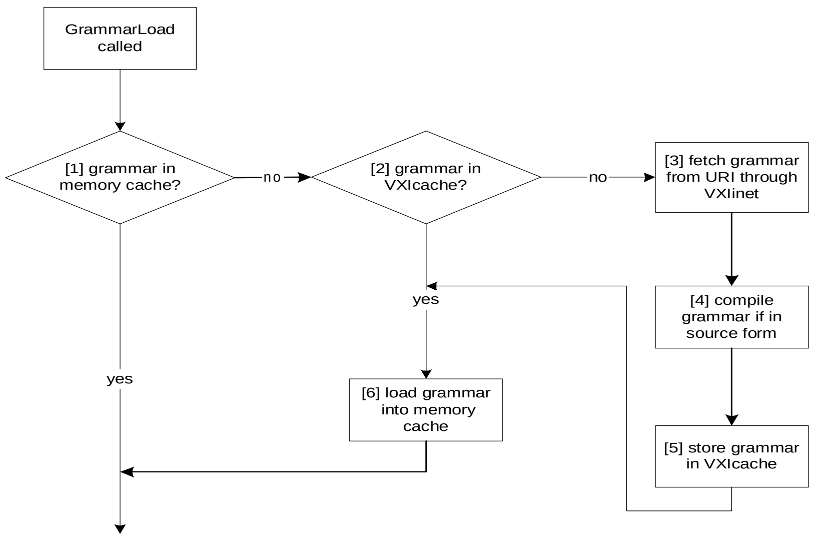
GrammarLoad called
[1] grammar in memory cache?
no
[2] grammar in VXIcache?
no
[3] fetch grammar from URI through VXIinet
yes
[4] compile grammar if in source form
yes
[6] load grammar into memory cache
[5] store grammar in VXIcache

Figure 3-4 Grammar fetching and caching

Figure 3-4 illustrates the flow of grammar fetching and caching from a call to load a grammar. The following section discusses specific VXIinet and VXIcache interface functions that are called by each of the above components. Convert the grammar URI (or source) into a hash key for Cache lookup.

The steps in the flow description are numbered to help you follow the flow.

1. Determine whether a grammar is in the memory cache.

```
if (entry is found in memory cache which matches URI hash)
  if (cache entry is valid [7])
    use cached entry
  else
    entry is not found
else
  entry is not found
```

2. Determine whether a grammar is in the VXIcache cache.

```
if (entry is present in disk cache [8])
  if (cache entry is valid [7])
    get cached entry by hashkey [9]
  else
    remove hash mapping from cache [10]
    entry is not found
else
  remove hash mapping from cache [10]
  entry is not found
```

3. Fetch grammar from URI through VXIinet [13].

4. Compile grammar if in source form. A dependency list is obtained of all grammars and dictionaries that are referenced. This dependency list is used during validation to ensure that all referenced components are also valid. For each URI of the grammar and its dependencies, the URI is fetched [13]. Some system level dependencies, such as dictionaries, are refetched if they are not modified.

5. Store grammar in VXIcache.

```
write to disk cache using hash key [11]
write hash key to disk cache using grammar ID [12]
```

6. Load grammar into memory cache.

```
read grammar from disk cache
convert grammar to in-memory form
add to memory cache using hash key
```

7. Determine whether a cache entry is valid.

```
look up all dependencies of cached entry
for specified URI and each dependency
  conditional fetch URI from VXIinet using validator [14]
  if (not valid)
    return not valid
return valid
```

8. Determine whether an entry is present in the disk cache.

```
  VXIcache->Open(id, CACHE_MODE_READ, flags, properties,
  stream_info):
   properties (passed in):
     CACHE_CREATION_COST --> CACHE_CREATION_COST_HIGH
   stream_info (expected back):
     CACHE_INFO_SIZE_BYTES --> size of cache entry
   flags: 0
VXIcache->Read();
VXIcache->Close();
```

9. Get cached entry by hash key.

```
  VXIcache->Open(hashkey, CACHE_MODE_READ, flags,
  properties, stream_info):
   properties (passed in):
     CACHE_CREATION_COST --> CACHE_CREATION_COST_HIGH
   stream_info (expected back):
     CACHE_INFO_SIZE_BYTES --> size of cache entry
   flags: 0
VXIcache->Read();
VXIcache->Close();
```

10. Remove hash mapping from cache.

```
  VXIcache->Open(id, CACHE_MODE_WRITE, flags, properties,
  stream_info):
   properties (passed in):
     CACHE_CREATION_COST --> CACHE_CREATION_COST_HIGH
   stream_info (expected back):
     nothing
   flags: 0
VXIcache->Write();
VXIcache->Close();
```

11. Write to disk cache using hash key.

```
  VXIcache->Open(hashkey, CACHE_MODE_WRITE, flags,
  properties, stream_info):
   properties (passed in):
     CACHE_CREATION_COST --> CACHE_CREATION_COST_HIGH
   stream_info (expected back):
     nothing
   flags: 0
VXIcache->Write();
VXIcache->Close();
```

12. Write hash key to disk cache using grammar ID.

```
VXIcache->Open(id, CACHE_MODE_WRITE, flags, properties,
stream_info):
 properties (passed in):
   CACHE_CREATION_COST --> CACHE_CREATION_COST_HIGH
 stream_info (expected back):
   nothing
 flags: 0
 VXIcache->Write();
 VXIcache->Close();
```

13. Fetch grammar URI.

```
VXIinet->Open(uri, INET_MODE_READ, flags, properties,
stream_info):
 properties (passed in):
   INET_URL_BASE --> dummy uri with current working
   directory so that relative local file references
   can be resolved
 stream_info (expected_back):
   INET_INFO_VALIDATOR --> validator associated with
   this URI
 flags: 0
 VXIinet->Read();
 VXIinet->Close();
```

14. Conditional fetch URI from VXIinet using validator.

```
VXIinet->Open(uri, INET_MODE_READ, properties, stream_
info):
 properties (passed in):
   INET_OPEN_IF_MODIFIED --> contains validator to check
   against
   INET_URL_BASE --> dummy uri with current working
   directory so that relative local file references
   can be resolved
 stream_info (expected_back):
   INET_INFO_VALIDATOR --> new validator associated with
   this URI
 VXIinet->Read();
 VXIinet->Close();
```

The steps described above are also used by OpenSpeech Browser to store audio that was generated from the text-to-speech system and in future releases for compiled forms of the VoiceXML documents. Thus the recognizer, prompting engine, and the interpreter all make use of the internet and cache components.

CHAPTER 4

OpenSpeech Browser API

This chapter describes OpenSpeech Browser API functions. Each description includes a list of the arguments (if any) and a sample piece of code using that function. The functions are listed by categories. Also included is a separate function list in alphabetic order.

API function categories

The following tables group the API functions by these categories:

- Application manager
- Hardware integration interface
- Internet interface
- Logging interface
- Object interface
- Prompting interface
- Recognition interface
- Telephony interface
- Thread library
- Types and value library
- VXI

Application manager	
VXIplatform	4-62
SBclient	4-66

Hardware integration interface	
VXIaudioplayer	4-68
SBaudioplayer	4-81
VXIaudiosource	4-83
SBaudiosource	4-93
VXIsessionControl	4-95
SBsessionControl	4-109

Internet interface	
VXIinet	4-112
SBinet	4-128
VXIcache	4-131
SBcache	4-135
VXIjsi	4-137
SBjsi	4-145

Alphabetical list of functions

[tbd - 9 - [?Which of these should be removed?]

Application manager

The application manager consists of a system operator user interface to start, stop, and configure the platform as well as a VXIplatform interface implementation that interacts with the OpenSpeech Browser components for initialization and servicing calls.

OpenSpeech Browser provides a basic application manager called testClient [?Not anymore], built on top of a reference VXIplatform interface implementation called SBclient. These components are provided in source code for integrators as a getting started toolkit. Integrators can use any part of this code in integrating the OpenSpeech Browser PIK into their platforms. See the src/client directory in the distribution.

- VXIplatform – VoiceXML platform interface
- SBclient – SBclient implementation of VXIplatform. [?Remove]

VXIplatform

Interface that supports multiple system operator user interfaces by initializing the OpenSpeech Browser browser components and coordinating them to service calls.

Function	Definition
VXIplatformCreateResources( )	Creates a VXIplatform object for the specified channel number.
VXIplatformDestroyResources( )	Destroys the VXIplatform and any associated resources.
VXIplatformEnableCall( )	Enables the hardware to wait for call.
VXIplatformInit( )	Performs initialization of the platform layer.
VXIplatformProcessDocument( )	Starts the processing of the root document associated with the channel.
VXIplatformResult	Result codes for the platform functions.
VXIplatformShutdown( )	Performs final cleanup of the platform layer.
VXIplatformWaitForCall( )	Waits for a call on the specified channel and answers the call.

VXIplatformResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Result code	Definition
VXIinet_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error
VXIplatform_RESULT_ALREADY_INITIALIZED	Platform already initialized
VXIplatform_RESULT_BUFFER_TOO_SMALL	Return buffer too small
VXIplatform_RESULT_FAILURE	Normal failure, nothing logged
VXIplatform_RESULT_FATAL_ERROR	Fatal error, terminate call
VXIplatform_RESULT_INET_ERROR	VXIinet interface error
VXIplatform_RESULT_INTERPRETER_ERROR	VXIinterpreter interface error
VXIplatform_RESULT_INVALID_ARGUMENT	Invalid function argument
VXIplatform_RESULT_INVALID_PROP_NAME	Property name is not valid
VXIplatform_RESULT_INVALID_PROP_VALUE	Property value is not valid
VXIplatform_RESULT_IO_ERROR	I/O error
VXIplatform_RESULT_JSI_ERROR	VXIjsi interface error
VXIplatform_RESULT_LOG_ERROR	XIlog interface error
VXIplatform_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error
VXIplatform_RESULT_NOT_INITIALIZED	Platform not initialized
VXIplatform_RESULT_OUT_OF_MEMORY	Out of memory
VXIplatform_RESULT_PLATFORM_ERROR	Errors from platform services
VXIplatform_RESULT_PROMPT_ERROR	VXIprompt interface error
VXIplatform_RESULT_REC_ERROR	VXIrec interface error
VXIplatform_RESULT_SUCCESS	Success
VXIplatform_RESULT_SYSTEM_ERROR	System error, out of service
VXIplatform_RESULT_TEL_ERROR	VXItel interface error
VXIplatform_RESULT_TRD_ERROR	VXItrd interface error
VXIplatform_RESULT_UNSUPPORTED	Operation is not supported
VXIplatform_RESULT_VALUE_ERROR	VXIvalue interface error

VXIplatformInit( )

```
VXIplatformResult VXIplatformInit(
  VXIMap* args,
  VXIunsigned * nbChannels
);
```

Parameter	Description
args	Configuration arguments. Implementation dependant.
nbChannels	The address of a pre-allocated VXIunsigned that is initialized with the number of available channels.

VXIplatformCreateResources( )

Creates a VXIplatform object for the specified channel number. This creates the VXItel, VXIprompt and VXIrec objects associated with this channel.

```
VXIplatformResult VXIplatformCreateResources(
  VXIunsigned channelNum,
  VXIMap * args,
  VXIplatform ** platform
);
```

Parameter	Description
channelNum	The channel number of the platform to be created.
args	Configuration arguments. Implementation dependant.
platform	A pointer to the platform to be allocated.

VXIplatformProcessDocument

```
VXIplatformResult VXIplatformProcessDocument(
  const VXIchar* url,
  VXIMap* sessionArgs,
  VXIValue** result,
  VXIplatform * platform
);
```

Parameter	Description
url	Name of the VoiceXML document to fetch and execute, may be a URL or a platform dependant path. See the Open( ) method in VXIinet.h for details about supported names, however for URLs this must always be an absolute URL and any query arguments must be embedded.
sessionArgs	Any arguments to be passed to the VXI. Some of these, such as ANI, DNIS, etc. as required by VXML, but anything may be passed in. These values are available through the session variable in ECMAScript.
result	(Optional, pass NULL if not desired.) Return value for the VoiceXML document (from), this is allocated on success and when there is an exit value (a NULL pointer is returned otherwise), the caller is responsible for destroying the returned value by calling VXIValueDestroy( ). If VXIinterp_RESULT_UNCAUGHT_FATAL_EVENT is returned, this is a VXIString that provides the name of the VoiceXML event that caused the interpreter to exit.
platform	A pointer to the platform to be allocated.

SBclient

[tbd - 10 - [?Remove this section?]

SBclient implementation of VXIplatform

VXIplatform

[tbd - 11 - [?Relevant?]

SBclient defined VXIplatform structure, returned upon initialization by VXIplatformCreateResources( ).

Provides an implementation of the VXIplatform abstract interface for creating and destroying the VXI resources required by the browser and to isolate the main from the actual implementation details of the resource management. The reference application manager uses this set of convenience functions.

Hardware integration interface

The OpenSpeech Browser uses three interfaces to communicate with the telephony hardware. These interfaces must be implemented by the platform in order for the OpenSpeech Browser to run on that specific hardware platform. In particular, the VXIsessionControl interface is the primary place where additional session control capabilities can be added into the browser.

The OpenSpeech Browser provides a reference implementation of these three interfaces for Dialogic CSP enabled hardware using Dialogic GlobalCall and Microsoft Visual Studio project files to compile the reference implementations. See the src/HW/Dialogic directory in the distribution, the results of which build into SBhw.dll.

The hardware integration interfaces are:

- VXIaudioplayer – Audio Player Interface.
- SBaudioplayer – SBaudioPlayer implementation of VXIaudioPlayer.
- VXIaudiosource – Audio Source Interface.
- SBaudiosource – SBaudioSource implementation of VXIaudioSource.
- VXIsessionControl – Session Control interface.
- SBsessionControl – SBsessionControl implementation of VXIsessionControl.

VXIaudioplayer

Abstract interface for playing audio to the caller via a telephony interface, where the audio may be on-disk audio files, in-memory audio, or in-memory streaming audio.

Feature	Description
VXIapResult	Result codes for interface methods.
VXIapPlayListNode	MIME content type for the data being passed.
AP_EVENT_STARTING	Events returned from the Audio Player object.
VXIapEventDataStatus	Event structure for returning detailed status information.
VXIapEventListener	Signature for the event handler called to report Audio Player events, as registered via RegisterEventListener( ).
VXIapInterface	Audio Player interface for audio playback.

The Audio Player interface must be implemented for each new underlying telephony interface, which is frequently done by a third party integrator.

There is one Audio Player interface per telephony channel.

This table summarizes the audio MIME content types that all Audio Player implementations should support (they support other MIME content types as well).

Content type	Description
audio/basic (VXI_MIME_ULAW)	Raw (headerless) 8kHz 8-bit mono u-law [PCM] single channel audio (G.711).
audio/x-alaw-basic (VXI_MIME_ALAW)	Raw (headerless) 8kHz 8-bit G.711 mono A-law [PCM] single channel audio (G.711).
audio/x-wav (VXI_MIME_WAV)	WAV (RIFF header) 8kHz 8-bit mono u-law or A-law [PCM] single channel audio.

This table summarizes the events delivered by Audio Player implementations:

Event	Description
AP_EVENT_COMPLETED	Reports that a play operation has terminated normally, either by the output operation successfully completing or the play operation being stopped via TriggerStop( ). Audio Player implementations can choose to automatically stop the play operation and return this event on hang-up. This should only be done when this is an automatic functionality within the underlying telephony library. (The client is responsible for rapidly calling TriggerStop( ) when barge-in or hang-up occurs, which in most cases simplifies the Audio Player implementation by avoiding session control and audio source interactions.) The Audio Player must move from the PLAYING or STREAMING state to an IDLE state prior to delivering this event.
AP_EVENT_ERROR	Reports the Audio Player encountered an error in playing a play node in the play list or a underlying telephony or I/O error. The error *must* have occurred after a call to VXIapTriggerPlay( ) or VXIapTriggerStreamingPlay( ) returned with a VXIap_RESULT_SUCCESS result code. The Audio Player must be returned to an IDLE state (the play operation halted) prior to delivering this event as the user may immediately trigger additional play operations from within the event listener. The event includes a result code that indicates the cause of the error and the playlist node that caused the error (when available).
AP_EVENT_RESUME_STREAM	Reports the Audio Player is ready to resume a streaming audio source that is currently paused, thus obtaining additional audio for the current streaming audio operation. The source was paused when an overflow occurred while delivering audio for the streaming play operation (TriggerStreamingPlay( ) returned VXIap_RESULT_STREAM_OVERFLOW).
AP_EVENT_STARTING	Reports the Audio Player moved from the IDLE state to the PLAYING or STREAMING state. Delivered only for the first play node in a play or streaming play operation.
AP_EVENT_WARNING	Reports the Audio Player encountered a non-fatal error in playing a play node in the play list or a underlying telephony or I/O error that is recoverable. The error *must* have occurred after a call to VXIapTriggerPlay( ) or VXIapTriggerStreamingPlay( ) returned with a VXIap_RESULT_SUCCESS result code. The warning has no implications on the current state, if the result of the warning is that the play operation completed an AP_EVENT_COMPLETED event which must be delivered after this event (or an AP_EVENT_ERROR delivered instead). The event includes a result code that indicates the cause of the error and the playlist node that caused the error (when available).

VXIapResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues)

Result code	Description
VXIap_RESULT_BAD_MIME_TYPE	Unsupported Mime type
VXIap_RESULT_BUFFER_TOO_SMALL	Return buffer too small
VXIap_RESULT_BUSY	Play operation in progress
VXIap_RESULT_DRIVER_ERROR	Low-level telephony library error
VXIap_RESULT_FAILURE	Normal failure, nothing logged
VXIap_RESULT_FATAL_ERROR	Fatal error, terminate call
VXIap_RESULT_FILE_ERROR	File open error
VXIap_RESULT_INVALID_ARGUMENT	Invalid function argument
VXIap_RESULT_INVALID_PARAM_NAME	Parameter name is not valid
VXIap_RESULT_INVALID_PARAM_VALUE	Parameter value is not valid
VXIap_RESULT_IO_ERROR	I/O error
VXIap_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error
VXIap_RESULT_NOTHING_TO_PLAY	There is nothing to play
VXIap_RESULT_OUT_OF_MEMORY	Out of memory
VXIap_RESULT_PLATFORM_ERROR	Errors from platform services
VXIap_RESULT_STREAM_OVERFLOW	The stream has overflowed
VXIap_RESULT_STREAM_UNDERFLOW	The stream has run out of data to play
VXIap_RESULT_SYSTEM_ERROR	System error, out of service
VXIap_RESULT_UNSUPPORTED	Operation is not supported

VXIapPlayListNode

Definition of a play list node, used to deliver a linked list of audio for playback.

```
VXIapPlayListNode {
  VXIchar *contentType;
  VXIchar *filePath;
  VXIptr content;
  VXIlong contentLen;
  struct VXIapPlayListNode *previous;
  struct VXIapPlayListNode *next;
  void (*Destroy)
};
```

Parameter	Description
contentType	MIME content type for the data being passed, even when the data is provided from a file. Audio Player implementations must use this, not file extension mapping rules, to determine the audio type for playback. The supported MIME content types are determined by the Audio Player implementation.
filePath	Full path to a file that contains the content to present. NULL if in-memory content is being supplied instead (is NULL for all streaming play operations).
content	In-memory content to present. NULL if a file path is being supplied instead.
contentLen	Length, in bytes, of the content to present. 0 if a file path is being supplied.
previous	Previous node in the play list. NULL if this is the head of the list. The Audio Player implementation can modify this for its own purposes whenever it accepts the node by returning VXIap_RESULT_SUCCESS from TriggerPlay( ) or TriggerStreamingPlay( ).
next	Next node in the play list. NULL if this is the end of the list. The Audio Player implementation can modify this for its own purposes whenever it accepts the node by returning VXIap_RESULT_SUCCESS from TriggerPlay( ) or TriggerStreamingPlay( ).
destroy	A destructor to destroy the node. The Audio Player implementation must call this destructor on each individual node in the linked list when it no longer requires the node, preferably as soon as the data is no longer required. Destruction of a node does NOT cause the destruction of the previous or next nodes.

AP_EVENT_STARTING

Events returned from the Audio Player as described above.

VXIapEventDataStatus

All Audio Player events return a status structure for providing detailed status information along with the affected play list node. This structure, along with all the data it points at, is only valid for the duration of the event listener invocation.

```
VXIapEventDataStatus {
  VXIapPlayListNode *playListNode;
  VXIapResult resultCode;
};
```

Parameter	Description
playListNode	Exact node playing when the event occurred, such as the node that caused an error condition for an error event. If the underlying telephony API prevents determining the exact node, this may be set to NULL.
resultCode	Result code indicating details for AP_EVENT_ERROR and AP_EVENT_WARNING events, typically set to VXIap_ RESULT_SUCCESS for other events.

Audio player events

Events returned from the Audio Player object.

Event	Description
AP_EVENT_COMPLETED	A play operation has terminated. This event is triggered regardless of how the play terminates. Termination may occur by the play operation running to completion, by the client calling TriggerStop, or by a far end hang-up or a telephony error. Audio Player moves from the PLAYING state, or the STREAMING state to the IDLE state with this event.
AP_EVENT_ERROR	Returned when the Audio Player encounters an error in playing a segment in the play list or a underlying I/O error. The error must have occurred after a call to VXIapTriggerPlay( ) or VXIapTriggerStreamingPlay( ) returned. The error returns the Audio Player to the IDLE state. The event includes a result code that indicates the cause of the error and the playlist node that caused the error if its available. The playlist node is only valid for the duration of the CallBack.
AP_EVENT_RESUME_STREAM	Delivered during a streaming play when the condition that caused the TriggerStreamingPlay to return VXIap_RESULT_STREAM_OVERFLOW is cleared. The application can resume calling TriggerStreamingPlay.
AP_EVENT_STARTING	Returned when the Audio Player moves from the IDLE state to the PLAYING or STREAMING state. Delivered only for the first play in a segment list or first call to stream.
AP_EVENT_WARNING	Returned when the Audio Player encounters a non-fatal error in playing a segment in the play list or a underlying I/O error that is recoverable. The error must have occurred after a call to VXIapTriggerPlay( ) or VXIapTriggerStreamingPlay( ) returned. The warning has no implications on the current state, if the result of the warning is that the play operation completed an AP_EVENT_COMPLETED event is delivered after this event. The event includes a result code that indicates the cause of the error and the playlist node that caused the error if its available. The playlist node is only valid for the duration of the CallBack.

VXIapEventListener

Signature for the event handler called to report Audio Player events, as registered via RegisterEventListener( ).

```
VXIapEventListener (
  VXIapInterface* pThis,
  VXIulong eventType,
  VXIapEventDataStatus* eventData,
  VXIptr userData
);
```

Parameter	Description
pThis	Audio player reporting the event.
eventType	Event that occurred.
eventData	Structure that reports the details of the event, where the structure and its members are only valid for the duration of the event listener call.
userData	User data set when the listener was registered.

VXIapInterface

Audio Player interface for audio playback.

Function	Definition
GetImplementationName( )	Get the name of the implementation.
GetOptimalAudioMimeType( )	Retrieve the preferred MIME content type for play operations.
GetOptimalStreamBufferSizeRange( )	Retrieve the preferred range of audio buffer sizes for streaming.
GetVersion( )	Get the VXI interface version implemented.
RegisterEventListener( )	Register an event listener/user data pair for event notification.
TriggerPlay( )	Triggers playing audio to the caller, non-blocking.
TriggerStop( )	Stops the play operation.
TriggerStreamingPlay( )	Triggers playing an audio stream to the caller, non-blocking.
UnregisterEventListener( )	Unregister an event listener/user data pair.

RegisterEventListener

Zero or more event listeners may be registered by calling this method, each with their own user data. The combination of the event listener function and user data pointer must be unique: if the same pair is registered twice, an error is returned.

```
VXIapResult (*RegisterEventListener) (
  struct VXIapInterface* pThis,
  VXIapEventListener* eventListener,
  const VXIptr userData
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
eventListener	Event listener function that is called when events occur.
userData	Opaque user defined data, passed to the listener for each listener invocation to permit the listener to access information it needs to react to the event.

UnregisterEventListener

Unsubscribes the given listener/userdata combination.

```
VXIapResult (*UnregisterEventListener) (
  struct VXIapInterface* pThis,
  VXIapEventListener* eventListener,
  const VXIptr userData
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
eventListener	Pointer to the function to be called to notifythe caller of events (see VXIapEventListener).
userData	Opaque data defined by the caller, passed into the registered event listener as user data.

GetOptimalAudioMimeType

This retrieves the preferred MIME content type for play operations, allowing clients to obtain that type whenever possible from audio data sources such as text-to-speech engines. Note: this value is only a hint since the client plays audio with other MIME content types such as when the application developer specifies an audio file.

```
VXIapResult (*GetOptimalAudioMimeType) (
  struct VXIapInterface* pThis,
  VXIchar* contentType,
  const VXIint contentTypeLen
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
contentType	MIME content type for the audio buffer.
contentTypeLen	contentType size, in characters.

GetOptimalStreamBufferSizeRange

This retrieves the preferred audio buffer size range for streaming play operations, allowing clients to obtain a buffer size within that range whenever possible from streaming audio data sources such as text-to-speech engines. Note: this value is only a hint since the client plays audio with other buffer sizes in cases where the desired range cannot be satisfied by the audio source.

```
VXIapResult (*GetOptimalStreamBufferSizeRange) (
  struct VXIapInterface* pThis,
  VXIulong* lowLimitBytes,
  VXIulong* highLimitBytes
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
lowLimit	The lower limit of the optimal range in bytes.
highLimit	The higher limit of the optimal range in bytes.

TriggerPlay

This routine is asynchronous, which means it only triggers the beginning of a play operation. TriggerPlay only supports one play operation at a time. If another play operation is initiated via TriggerPlay( ) or TriggerStreamingPlay( ) before this play operation completes, a VXIap_RESULT_BUSY error is returned.

When TriggerPlay returns successfully, ownership of the entire play list is passed to the Audio Player, which may then modify the previous and next pointers in the play list to support the implementation. The Audio Player is responsible for calling the Destroy( ) method of each play node that is no longer needed. If TriggerPlay returns an error, however, this function must not have made any changes to the play list (particularly the previous and next pointers), and ownership is returned to the client.

```
VXIapResult (*TriggerPlay) (
  struct VXIapInterface* pThis,
  VXIapPlayListNode* playList,
  const VXIint playListSize,
  VXIapPlayListNode** invalidNode
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
playList	List of play nodes to play to the caller, on success ownership is passed.
playListSize	Number of nodes in the play list.
invalidNode	On error, this is set to point at the node in the play list that triggered the error, or NULL if there is a generic error.

TriggerStop

This routine is synchronous and returns when the play is either stopped or there is an error. An AP_EVENT_COMPLETED event must be delivered before this returns.

```
VXIapResult (*TriggerStop) (
  struct VXIapInterface* pThis
);
```

Parameter	Description
pThis	Pointer to the instance of the service.

TriggerStreamingPlay

This routine is asynchronous, in that it only triggers the beginning of a play operation. TriggerStreamingPlay only supports one play operation at a time. However, for streaming playback this function is called multiple times by the client in order to provide the streaming audio as it becomes available from the streaming source. If another play operation is initiated via TriggerPlay( ) before this operation completes, a VXIap_RESULT_BUSY error is returned.

The first call to TriggerStreamingPlay( ) (while the Audio Player is in an IDLE state) initiates a streaming play operation. For streaming plays, every play node always contains in-memory audio, never a pointer to a file. The MIME content type and number of bytes of audio within each play node do not vary within a single streaming play operation.

The client delivers streaming audio as fast as possible, making no attempt to recover from network delays (jitter) during play operations, so the Audio Player should pre-buffer an implementation defined quantity of audio before actually starting playback to avoid underruns. The AP_EVENT_STARTING event must only be delivered once the actual audio starts getting heard by the caller (playback actually starts). If an underflow does occur, the Audio Player must deliver an AP_ EVENT_WARNING event with a XIap_RESULT_STREAM_UNDERFLOW status code to notify the client, then re-build the pre-buffer before resuming playback again.

For flow control, the Audio Player should maintain high and low watermark thresholds. Whenever the amount of buffered audio is less then the implementation defined high watermark, accept audio from TriggerStreamingPlay( ) calls. When it's exceeded, return a VXIap_RESULT_STREAM_OVERFLOW error to instruct the client to suppress additional audio until the Audio Player delivers an AP_ EVENT_ RESUME_STREAM event. The Audio Player should only do so when the amount of buffered audio falls below the low watermark. This mechanism, where there is some distance between the low and high watermarks, ensures the flow control is done efficiently by minimizing the amount of times the audio stream is paused and resumed, as those operations are relatively expensive.

When this function returns successfully, ownership of the play node is passed to the Audio Player, which may then modify the previous and next pointers to support the implementation. The Audio Player is responsible for calling the Destroy( ) method of each play node that is no longer needed. If TriggerStreamingPlay returns an error, however, this function must not have made any changes to the play node (particularly the previous and next pointers), and ownership is returned to the client.

```
VXIapResult (*TriggerStreamingPlay) (
 struct VXIapInterface*    pThis,
 VXIapPlayListNode*        playNode,
 const VXIbool             lastPlayInStream
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
playNode	Node to play to the caller, on success ownership is passed.
lastPlayInStream	TRUE to indicate this is the last play node in the stream, FALSE to indicate additional play nodes follow.

SBaudioplayer

Provides a Dialogic CSP implementation of the VXIaudioPlayer abstract interface for telephony hardware audio playback.

There is one audio player interface per thread/line.

Function	Definition
SBaudioPlayerCreateResource	Creates a new audio player resource and returns its handle.
SBaudioPlayerDestroyResource	Destroys an audio player resource.

SBaudioPlayerCreateResource

This routine creates a new instance of a dialogic/JCT implementation of an audio player resource. The resource is linked to a physical channel through the channelDev which must be a voice device obtained from a previously constructed session control resource through its SBsessionControlGetVoiceDevice routine.

```
VXIapResult SBaudioPlayerCreateResource (
  VXIapInterface **pThis,
  VXIscInterface* session
);
```

Parameter	Description
pThis	Pointer to the newly created instance of the VXIapInterface.
session	A previously created session control resource, the created audio player resource acts on the channel associated to the session control resource.

SBaudioPlayerDestroyResource

This routine destroys an instance of a dialogic/JCT implementation of an audio player resource. Upon successful return pThis is NULLed out.

```
VXIapResult SBaudioPlayerDestroyResource (
  VXIapInterface **pThis
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIapInterface that is to be destroyed.

VXIaudiosource

Abstract interface for obtaining audio (speech) from the caller via a telephony interface, where the audio is provided as in-memory streaming audio delivered via events.

The Audio Source interface must be implemented for each new underlying telephony interface, which is frequently done by a third party integrator.

Function	Definition
VXIasResult	Result codes for interface methods.
VXIasCallbackResult	Result codes for event listener callbacks.
AS_EVENT_ERROR	Events returned from the Audio Source.
VXIasEventData	Event structure for returning audio and/or detailed status information.
VXIasEventListener	Signature for the event handler called to report Audio Source events, as registered via RegisterEventListener( ).
VXIasInterface	Audio Source interface for audio input.

The following table summarizes the events delivered by Audio Source implementations.

Function	Description
AS_EVENT_ERROR	Reports the Audio Source encountered an error in delivering audio during a record operation, typically due to an underlying telephony or I/O error. The error must have occurred after a call to VXIasTriggerRecord( ) returned with a VXIas_RESULT_SUCCESS result code, never during that call or after that call returned with an error. The Audio Source must be returned to an IDLE state (the record operation halted) prior to delivering this event.
AS_EVENT_RECEIVED	Reports audio (possibly caller speech) that has been received from the telephony interface. This audio should be echo-cancelled audio (done by the telephony interface or an external device). It can be from a first-pass speech detector, which uses the telephony interface or an external device to suppress audio delivery until potential speech is identified within the audio stream (either a noise detector or more intelligent speech detector). However, if a first-pass speech detector is used (most commonly to reduce CPU consumption by avoiding streaming silence that occurs prior to speech), after it detects potential speech it must deliver audio starting at 250 msec BEFORE the speech detector fired. This allows a higher-level, more sophisticated endpointer integrated above this interface to properly scan for low-energy speech that first-pass speech detectors commonly LEAVE off.
AS_EVENT_STOPPED	Reports that a record operation has terminated normally, either due to the input operation successfully completing or the record operation being stopped via TriggerStop( ). Audio Source implementations may also choose to automatically stop the record operation and return this event on hang-up. This should only be done when this is an automatic functionality within the underlying telephony library. (The client is responsible for rapidly calling TriggerStop( ) when hang-up occurs, which in most cases simplifies the Audio Source implementation by avoiding session control interactions.) The Audio Source must move from the RECORDING state to an IDLE state prior to delivering this event.
AS_EVENT_OVERFLOW	Reports the Audio Source has overflowed its internal buffers while in a PAUSED state and is throwing out audio (the implementation decides whether to discard audio first-in-first-out or last-in-first-out). This is not reported again until the record operation is resumed and then paused again. The record operation otherwise proceeds normally once resumed. The PAUSE state was entered due to the event listener returning VXIasCb_RESULT_ PAUSE_AUDIO for an AS_EVENT_RECEIVED event.

VXIasResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Function	Description
VXIas_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIas_RESULT_BUSY	Play operation in progress.
VXIas_RESULT_DRIVER_ERROR	Low-level telephony library error.
VXIas_RESULT_IO_ERROR	I/O error.
VXIas_RESULT_OUT_OF_MEMORY	Out of memory.
VXIas_RESULT_SYSTEM_ERROR	System error, out of service.
VXIas_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIas_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIas_RESULT_INVALID_PARAM_NAME	Parameter name is not valid.
VXIas_RESULT_INVALID_PARAM_VALUE	Parameter value is not valid.
VXIas_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIas_RESULT_FAILURE	Normal failure, nothing logged.
VXIas_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIas_RESULT_BAD_MIME_TYPE	Unsupported MIME type.
VXIas_RESULT_UNSUPPORTED	Operation is not supported.

VXIasCallbackResult

Result codes for event listener callbacks.

Function	Description
VXIasCb_RESULT_SUCCESS	Event listener callback was successful.
VXIasCb_RESULT_PAUSE_AUDIO	Event listener callback requests pausing the delivery of additional audio until TriggerResume( ) is called, as the consumer is not ready for additional audio. The Audio Source may opt to ignore this (it should log an error to the operator console if doing so), or may support this by buffering the audio, AS_EVENT_ERROR, and AS_EVENT_STOPPED events while delivering an AS_EVENT_OVERFLOW event if the Audio Source audio buffer overflows, and delivering all the buffered audio and events in the normal order if TriggerResume( ) is called.

Audio source events

Events returned from the Audio Source object.

Function	Description
AP_EVENT_ERROR	Returned when the Audio Source encounters an error in recording. The error returns the Audio Source to the IDLE state.
AP_EVENT_RECEIVED	This event is used to deliver the audio to all event listeners. This is the way that audio is transferred from the hardware.
AP_EVENT_STOPPED	Delivered whenever the audio source is stopped. This should be delivered no matter how the audio source is stopped.
AP_EVENT_OVERFLOW	Delivered when the interface overflows its internal buffer and is now dropping audio. The implementer determines if the audio is dropped first-in-first-out or last-in-last-out. Audio must always be delivered first-in-first-out.

AS_EVENT_ERROR

Events returned from the Audio Source as detailed in the Audio Source interface description.

VXIasEventData

Event structure for returning audio and/or detailed status information.

All Audio Source events return an event data structure for providing audio from the record operation and/or detailed status information. This structure, along with all the data it points at, is only valid for the duration of the event listener invocation.

```
VXIapEventData (
  VXIulong dataSizeBytes,
  VXIptr data,
  VXIchar* mimetype,
  VXIulong sequenceNum,
  VXIasResult resultCode
);
```

Function	Description
dataSizeBytes	Size of the audio data in bytes, may be 0.
data	Pointer to the audio data, NULL if dataSizeBytes is 0.

Function	Description
mimetype	MIME content type for the audio data, must match an audio type supported by the recognizer. Typical choices are "audio/basic" (VXI_MIME_ULAW, Raw (headerless) 8kHz 8-bit mono u-law [PCM] single channel audio [G.711]) and "audio/x-alaw-basic" (VXI_MIME_ALAW, Raw (headerless) 8kHz 8-bit mono A-law [PCM] single channel audio [G.711]).
sequenceNum	When dataSizeBytes is greater then 0 (audio data is being delivered), sequence number of the data packet, where the first audio packet after TriggerRecord( ) is 1, and then each packet afterwards increments the sequence number by one. (A TriggerStop( ) operation followed by a TriggerRecord( ) thus resets the sequence number to 1.) If data packets are discarded due to buffer overflows (when an AS_EVENT_OVERFLOW is delivered), the sequence number should also be incremented by one for each packet discarded.
resultCode	Result code indicating details for AS_EVENT_ERROR events, typically set to VXIas_RESULT_SUCCESS for other events.

When the audio source is in the Recording state, VXIasEventData events are returned to all registered event listeners. Sequence numbers in the events must increase and start at 1.

The mime type of each event must match the mime type of the first event. The mime type information is provided as is to the OpenSpeech Recognizer. See the *OpenSpeech Recognizer* manual for a list of acceptable mime types. If the mime type is NULL, a mime type of audio/basic (u-law) is assumed.

The dataSizeBytes may vary from event to event. The recommended setting size is between 20 and 200 ms of audio. The OpenSpeech Recognizer recommends that less than 4000 bytes be delivered in each call.

VXIasEventListener( )

As a flow control mechanism, this may return VXIasCb_RESULT_PAUSE_AUDIO to request the Audio Source to suspend the delivery of audio packets, instead of buffering the audio. Once the consumer is ready for more audio, the audio packet delivery is resumed by calling TriggerResume( ).

NOTE

This Audio Source functionality is optional, and Audio Source implementations that support this may have limited buffer sizes and thus may return an AS_EVENT_OVERFLOW error (throwing out audio warning) if the stream is paused for long.

```
typedef VXIasCallbackResult (
  struct VXIasInterface* pThis,
  const VXIulong eventType,
  const VXIasEventData* eventData,
  const VXIptr userData
):
```

Parameter	Description
pThis	Audio source reporting the event
eventType	Type of the event that occurred.
eventData	VXIasEventDataStatus structure that reports the details of the event, where the structure and its members are only valid for the duration of the event listener call (audio must be deep copied if required for a longer time period).
userData	User data for the listener, as specified when registering the listener.

VXIasInterface

The AudioSource interface for audio input.

Function	Description
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
RegisterEventListener	Register an event listener/user data pair for event notification.
UnregisterEventListener	Unregister an event listener/user data pair.
TriggerRecord	Triggers delivery of non-blocking audio input data to all listeners.
TriggerStop	Stops the record operation.
TriggerResume	Resumes the delivery of audio input data.

RegisterEventListener

Zero or more event listeners may be registered by calling this method, each with their own user data. The combination of the event listener function and user data pointer must be unique: if the same pair is registered twice, an error is returned.

```
VXIasResult (*RegisterEventListener) (
  struct VXIasInterface* pThis,
  VXIasEventListener* eventListener,
  const VXIptr userData
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIasInterface service.
eventListener	Event listener function that is called when events occur.
userData	Opaque user defined data, passed to the listener for each listener invocation to permit the listener to access information it needs to react to the event.

UnregisterEventListener

Unsubscribes the given listener/userdata combination.

```
VXIasResult (*UnregisterEventListener) (
 struct VXIasInterface*      pThis,
 VXIasEventListener*      eventListener,
 const VXIptr      userData
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIasInterface service.
eventListener	Pointer to the function that is called to notify the caller that audio is available.
userData	Opaque data defined by the caller, passed into the registered event listener as the userData pointer for each event notification.

TriggerRecord

This only triggers audio input data delivery. This routine is expected to return after data delivery has been enabled (as opposed to blocking until data delivery is complete).

```
VXIasResult TriggerRecord (
 struct VXIasInterface*          pThis,
 const VXIchar*          mimeType
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIasInterface service.
mimeType	MIME content type for the audio input data, may be NULL or an empty string to indicate that any MIME content type supported by the recognizer is acceptable.

TriggerStop

This routine is synchronous and returns when the record operation is either stopped or there is an error. An AS_EVENT_STOPPED event must be delivered before this returns.

```
VXIasResult TriggerStop (
 struct VXIasInterface*        pThis,
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIasInterface service.

TriggerResume

If an event listener returns a VXIasCb_RESULT_PAUSE_AUDIO return code, the Audio Source is requested to pause audio delivery. This call resumes audio delivery, where the buffered audio is rapidly delivered until either it “catches up” with the real time audio input feed, or until an event listener pauses delivery again. (If the Audio Source audio buffer overflows, an AS_EVENT_OVERFLOW event is delivered.)

However, responding to VXIasCb_RESULT_PAUSE_AUDIO is optional. If the audio source cannot support that functionality, all calls to this should return VXIas_RESULT_UNSUPPORTED for symmetry.

```
VXIasResult TriggerResume (
 struct VXIasInterface*     pThis,
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIasInterface service.

SBaudiosource

Provides a Dialogic CSP implementation of the VXIaudioSource abstract interface for telephony hardware audio recording.

There is one audio source interface per thread/line.

Function	Definition
SBaudioSourceCreateResource	Creates a new audio source resource and returns its handle.
SBaudioSourceDestroyResource	Destroys an audio source resource.

SBaudioSourceCreateResource

This routine creates a new instance of a dialogic/JCT implementation of an audio source resource. The resource is linked to a physical channel through the channelDev which must be a voice device obtained from a previously constructed session control resource through its SBsessionControlGetVoiceDevice routine.

```
VXIapResult SBaudioSourceCreateResource (
 VXIasInterface   **pThis,
 VXIscInterface*   session
);
```

Parameter	Description
pThis	Pointer to the newly created instance of the VXIasInterface.
session	A previously created session control resource, the created audio source resource acts on the channel associated to the session control resource.

SBaudioSourceDestroyResource

This routine destroys an instance of a dialogic/JCT implementation of an audio source resource. Upon successful return pThis is NULLed out.

```
VXIapResult SBaudioSourceDestroyResource (
 VXIasInterface   **pThis
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIasInterface that is to be destroyed.

VXIsessionControl

Abstract interface for call control and obtaining DTMF input from the caller via a telephony interface. The call control functionality includes detecting incoming calls, accepting incoming calls, near-end disconnects (hang-up), detecting far-end disconnects, and performing call transfers.

Function	Description
VXIscResult	Result codes for interface methods.
VXIscTransferType	Types of transfers that can be performed.
SC_ANI_CALLER_NUMBER	Keys identifying standard read-only properties returned by GetNthPropertyPair( ). Support for these is optional but is highly recommended.
SC_XFER_CONNECT_TIMEOUT	Keys identifying standard read/write properties that are set using SetProperty( ).
SC_ANI_CALLER_NUMBER_DEFAULT	Defaults for standard properties.
SC_EVENT_ERROR	Events returned from the Session Control as detailed in the Session Control interface description.
VXIscEventData	Event structure for returning detailed status information.
VXIscEventListener	Signature for the event handler called to report Session Control events, as registered via RegisterEventListener( ).
VXIscInterface	Session Control interface for call control and DTMF input.

The Session Control interface must be implemented for each new underlying telephony interface, which is frequently done by a third party integrator.

There is one Session Control interface per telephony channel.

The following table summarizes the events delivered by Session Control implementations.

Function	Description
SC_EVENT_ERROR	Reports the Session Control encountered an error on the line during a call, typically due to an underlying telephony error. The error must have occurred after a call to VXIscAcceptSession( ) returned with a VXIsc_ RESULT_SUCCESS result code, never within that function or after that function returned with an error. If the error terminated the call, a SC_ EVENT_END event must be delivered after this event. The event includes a result code that indicates the cause of the error.
SC_EVENT_BEGIN	Reports a call has started, as initiated by a successful call to AcceptSession( ).
SC_EVENT_END	Reports a disconnect (hang-up, call terminated) due to either a near-end disconnect (from a call to TerminateSession( )), a far-end disconnect, or an error (in which case a SC_EVENT_ERROR is delivered prior to this event).
SC_EVENT_TRANSFER_BEGIN	Reports that a transfer has successfully started, as initiated by a successful call to Transfer( ). For a bridge (supervised) transfer, this event is reported when the far-end has answered the call and the near-end party and far-end party are able to hear each other. For a blind transfer this is reported when the blind transfer has been successfully initiated in the network. For a blind transfer, a SC_EVENT_END event should immediately follow this event.
SC_EVENT_TRANSFER_COMPLETE	Reports that a bridge (supervised) transfer has completed with the far-end party disconnected and the near-end party still connected. This can occur either on a maxtime exceeded or far-end party or network disconnect.
SC_EVENT_INVITE	Reports an incoming call request on the telephony channel. For analog connections, it is generated on each ring.
SC_EVENT_DTMF_RECEIVED	Reports a DTMF key press, with a separate event for each DTMF received. If no event listeners are registered, these events are buffered and delivered as soon as a listener is registered.

VXIscResult

Result codes for interface methods.

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Function	Description
VXIsc_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIsc_RESULT_INVALID_PROPERTY	The specified property does not exist.
VXIsc_RESULT_BUSY	Session in progress.
VXIsc_RESULT_DRIVER_ERROR	Low-level telephony library error.
VXIsc_RESULT_IO_ERROR	I/O error.
VXIsc_RESULT_OUT_OF_MEMORY	Out of memory.
VXIsc_RESULT_SYSTEM_ERROR	System error, out of service.
VXIsc_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIsc_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIsc_RESULT_INVALID_PARAM_NAME	Parameter name is not valid.
VXIsc_RESULT_INVALID_PARAM_VALUE	Parameter value is not valid.
VXIsc_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIsc_RESULT_FAILURE	Normal failure, nothing logged.
VXIsc_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIsc_RESULT_NO_DTMF	Non-fatal non-specific error.
VXIsc_RESULT_NEAR_END_DISCONNECT	Call terminated by a local hangup.
VXIsc_RESULT_FAR_END_DISCONNECT	Call terminated by a remote hangup.
VXIsc_RESULT_NETWORK_DISCONNECT	Call terminated by a network hangup.
VXIsc_RESULT_NOANSWER	Call failed because there was no answer.
VXIsc_RESULT_FAR_END_BUSY	Called party was busy.
VXIsc_RESULT_NETWORK_BUSY	Network party was busy.
VXIsc_RESULT_TIME_EXCEEDED	Transfer has exceeded its maximum allowed time.
VXIsc_RESULT_UNSUPPORTED	Operation is not supported.

VXIscTransferType

Types of transfers that can be performed.

SC_ANI_CALLER_NUMBER

Keys identifying standard read-only properties returned by GetNthPropertyPair( ). Support for these is optional but is highly recommended. Implementations may support other read-only properties as well.

SC_XFER_CONNECT_TIMEOUT

Keys identifying standard read/write properties that are set using SetProperty( ). Support for these is required when Transfer( ) is implemented, all time durations are in milliseconds. Implementations may support other read/write properties as well.

SC_ANI_CALLER_NUMBER_DEFAULT

Defaults for standard properties.

SC_EVENT_ERROR

Events returned from the Session Control as detailed in the Session Control interface description.

VXIscEventData

Event structure for returning detailed status information

All Session Control events return a status structure for providing detailed session and/or status information. This structure, along with all the data it points at, is only valid for the duration of the event listener invocation.

```
VXIscEventData   (
 VXIulong  dataSizeBytes,
 VXIptr  data,
 VXIchar*  mimetype,
 VXIscResult  resultCode
);
```

Parameter	Description
dataSizeBytes	Size of the data in bytes, may be 0.
data	Pointer to the data, NULL if dataSizeBytes is 0.
mimetype	MIME content type of the data.
resultCode	Result code indicating details for SC_EVENT_ERROR events, typically set to VXIsc_RESULT_SUCCESS for other events.

VXIscEventListener

Signature for the event handler called to report Session Control events, as registered via RegisterEventListener( ).

```
VXIscEventListener   (
 VXIscInterface*    pThis,
 VXIulong    eventType,
 const VXIscEventData *  eventData,
 const VXIptr    userData
);
```

Parameter	Description
pThis	Session control reporting the event.
eventType	Event type that occurred.
eventData	VXIscEventDataStatus structure that reports the details of the event, where the structure and its members are only valid for the duration of the event listener call.
userData	User data for the listener, as specified when registering the listener.

SC_XFER_MAXTIME

Keys identifying standard properties that are set using SetProperty( ).

Session control events

Events returned from the Session Control object.

Function	Description
SC_EVENT_ERROR	Event generated when an error occurs on the line. The line may be returned to idle by this event. In this case an additional END event is thrown.
SC_EVENT_END	Event generated when the in-bound line is terminated.
SC_EVENT_BEGIN	Event generated when the phone is being picked up on the completion of a call to AcceptSession.
SC_EVENT_TRANSFER_BEGIN	Event generated when the transfer is begun. In a bridge transfer, this event should be thrown when the call bridge is set-up. In a blind transfer, this should be set-up when the blind transfer departs the system.
SC_EVENT_TRANSFER_COMPLETE	Event generated when a bridge transfer is completed by having the far party disconnected. This occurs when a maxtime exceeded or far party or network disconnect on the far line.
SC_EVENT_TRANSFER_INVITE	Event generated when a ring or an equivalent VoIP message is received. Should be generated on each ring.
SC_EVENT_DTMF_RECEIVED	Event generated when a DTMF is received. Sent for each DTMF received. If a DTMF is already in the queue for which an event has not been generated, this event should be sent when the listener is enabled.

VXIscSessionData

Structure to deliver line event data.

Function	Description
dataSizeBytes	Size of returned event data in bytes.
data	The Event data.
mimetype	Type of data.
resultCode	The event result.

VXIscInterface

Session Control interface for call control and DTMF input.

Function	Description
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
RegisterEventListener	Register an event listener/user data pair for event notification.
UnregisterEventListener	Unregister an event listener/user data pair.
SetProperty	Set a property associated with the caller session.
GetNthPropertyPair	Gets the Nth property pair.
WaitForCall	Re-arm the telephony interface for new calls.
AcceptSession	Used to accept an invite for a new call.
TerminateSession	Terminate (hang-up) the call.
Transfer	Transfers the calling party to another party.
GetDTMF	Read a single DTMF digit from the DTMF queue.
FlushDTMFQueue	Remove all DTMF digits from the DTMF queue.

RegisterEventListener

Zero or more event listeners may be registered by calling this method, each with their own user data. The combination of the event listener function and user data pointer must be unique: if the same pair is registered twice, an error is returned.

```
VXIscResult (*RegisterEventListener) (
 struct VXIscInterface*        pThis,
 VXIscEventListener*           eventListener,
 const VXIptr                  userData
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIscInterface.
eventListener	Event listener function that is called when events occur.
userData	Opaque user defined data, passed to the listener for each listener invocation to permit the listener to access information it needs to react to the event.

UnregisterEventListener

Unsubscribes the given listener/userdata combination.

```
VXIscResult (*UnregisterEventListener) (
 struct VXIscInterface*     pThis,
 VXIscEventListener*     eventListener,
 const VXIptr     userData
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIscInterface.
eventListener	Pointer to the function that is called to notify the caller that audio is available.
userData	Opaque data defined by the caller, passed into the registered event listener as the userData pointer for each event notification.

SetProperty

See the SC_[...] properties as defined above for the standard properties, each implementation may define additional properties as well.

```
VXIscResult (*SetProperty) (
 struct VXIscInterface*    pThis,
 const VXIchar*    propertyName,
 const VXIunsigned    valueBuffSizeBytes,
 const VXIptr    value
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
propertyName	Name of property to set.
valueBuffSizeBytes	Size of value, in bytes. For the standard properties, pass wcslen(valueStr) * sizeof(wchar_t).
value	Value for the property to set. Note this is an untyped pointer, it is up to the Session Control implementation to define the appropriate type for each property (all the standard properties are VXIchar * based).

GetNthPropertyPair

Used to iterate through the Session Control properties, retrieving each key/value pair. Most commonly used to retrieve all the properties in order to expose them for application use, allowing the Session Control implementation to expose implementation specific properties to the application without requiring voice processing platform changes.

```
VXIscResult (*GetNthPropertyPair) (
 struct VXIscInterface*    pThis,
 const VXIunsigned    index,
 VXIunsigned*    nameBuffSizeChars,
 VXIchar*    propertyName,
 VXIunsigned*    valueBuffSizeChars,
 VXIchar*    value
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
index	Property index to retrieve, where the first property is index 0.
nameBuffSizeChars	As input, the size of the name buffer, in characters. This is then modified to return the actual name size, in bytes. When the name buffer is too small, this modified name can be used to allocate a larger buffer and re-try.
propertyName	Name of property to retrieve.
valueBuffSizeChars	As input, the size of the value buffer, in characters. This is then modified to return the actual value size, in bytes. When the value buffer is too small, this modified value can be used to allocate a larger buffer and re-try.
value	Value for the property.

WaitForCall

This call is blocking. When it returns the telephony interface must be ready to accept calls and generate events.

The hardware interface must start-up in an out-of-service (OOS) state. In this state, no events are generated and calls should be blocked (if possible) or rejected. The best implementation is to arrange for the line to be out of service back to the switch so that fail over to the next line occurs at the switch level.For some protocols setting the line to a busy state can do this.

A hang-up or error on the line should place the hardware back into the OOS state so that no calls come into the line until it is explicitly re-armed by calling this function again.

```
VXIscResult (*WaitForCall) (
 struct VXIscInterface*    pThis,
);
```

Parameter	Description
pThis	Pointer to the instance of the service.

AcceptSession

After a SC_EVENT_INVITE, this is called to accept the session, connecting the caller and starting the call. Some Session Control implementations may support (or require) passing implementation defined data when accepting a session.

```
VXIscResult (*AcceptSession) (
 struct VXIscInterface*    pThis,
 const VXIchar*    mimeType,
 const VXIunsigned*    dataSizeBytes,
 const VXIptr*    data
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
mimeType	MIME content type of the implementation defined data, typically NULL.
dataSizeBytes	Size of the implementation defined data, in bytes, typically 0.
data	Implementation defined data, typically NULL.

TerminateSession

Ends the current session. The caller is responsible for stopping the Audio Player and Audio Source interfaces if they are active.

```
VXIscResult (*TerminateSession) (
 struct VXIscInterface*    pThis,
);
```

Parameter	Description
pThis	Pointer to the instance of the service.

Transfer

This call is blocking. Two types of transfers are supported: bridge (supervised) and blind (unsupervised). For both, a SC_EVENT_TRANSFER_BEGIN event is delivered when the transfer is successfully initiated.

For blind transfers, this is followed by a SC_EVENT_END, as the transfer operation terminates the session. There is no guarantee that the far-end party actually answers the call when performing a blind transfer.

For bridging transfers, the transfer is fully supervised, with the session being retained in a non-interactive state until the transfer completes. If the near-end party disconnects, a SC_EVENT_END is delivered, and the session terminates. If the far-end party disconnects normally or is disconnected (due to exceeding the maximum time limit), or the transfer failed (due to a no answer, network failure, etc.) a SC_ EVENT_TRANSFER_COMPLETE event is delivered, this function returns, and the interactive session may proceed.

The SC_XFER_CONNECT_TIMEOUT property controls the amount of time allowed for a transfer connect attempt before the transfer is cancelled and the interactive session restored.

The SC_XFER_MAXTIME property controls the maximum time that the near-end and far-end parties may talk once they are connected. When this is exceeded, the far-end party is disconnected and the interactive session is restored.

Some Session Control implementations may support (or require) passing implementation defined data when performing a transfer.

```
VXIscResult (*Transfer) (
 struct VXIscInterface*    pThis,
 const VXIchar*    transferDest,
 const VXIchar*    mimeType,
 const VXIunsigned    dataSizeBytes,
 const VXIscTransferType    transferType,
 const VXIptr    data
);
```

Parameter	Description
pThis	Pointer to the instance of the service.
transferDest	URI that identifies the transfer destination. Must support the tel: URI syntax as defined in RFC 2806 (http:www.ietf.org/rfc/rfc2806.txt), implementations can opt to support other URI formats as well.
mimeType	MIME content type of the implementation defined data, typically NULL.
dataSizeBytes	Size of the implementation defined data, in bytes, typically 0.
transferTypee	Type of transfer, VXIscXferType_BLIND or VXIscXferType_ BRIDGE.
data	Implementation defined data, typically NULL.

GetDTMF

```
VXIscResult (*GetDTMF)(
 struct VXIscInterface*    pThis,
 VXIbyte*    dtmfDigit
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIscInterface.
ttbudtmfDigit	Returns the DTMF digit, returned as an ASCII character (0 - 9, *, #, A - D).

FlushDTMFQueue

There is only one DTMF queue and it is only accessible via this routine and GetDTMF( ). In systems with multiple clients, they must negotiate among themselves for access to the queue.

```
VXIscResult (*Flush) (
 struct VXIscInterface*    pThis,
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIscInterface.

SBsessionControl

Provides a Dialogic GlobalCall implementation of the VXIsessionControl abstract interface for telephony hardware call control.

There is one session control interface per thread/line.

Function	Definition
SBsessionControlCreateResource	Creates a new session control resource and returns its handle.
SBsessionControlDestroyResource	Destroy a session control resource.
SBsessionControlGetNumChannels	Gets the total number of dialogic channels installed on the machine.

SBsessionControlCreateResource

This routine creates a new instance of a dialogic/GlobalCall implementation of a session control resource. The resource is associated to the physical channel specified by the channelNum. Physical channels are 0 based and are contiguous across all dialogic boards to the total number of physical dialogic channels installed: see "SBsessionControlGetNumChannels".

Global call requires that when opening a channel a protocol is specified which drives the telephony standard for the channel. The protocol for the dialogic card must be obtained from dialogic and installed prior to running this implementation.

```
VXIscResult SBsessionControlCreateResource (
 VXIscInterface   **pThis,
 int   channelNum,
 const char*   gcProtocol
);
```

Parameter	Description
pThis	Pointer to the newly created instance of the VXIscInterface.
channelNum	The physical channel to associate the resource to.
gcProtocol	The name of the GlobalCall protocol for the channel.

SBsessionControlDestroyResource

This routine destroys an instance of a dialogic/GlobalCall implementation of a session control resource. Upon successful return pThis is NULLed out.

```
VXIscResult SBsessionControlDestroyResource (
 VXIscInterface   **pThis
);
```

Parameter	Description
pThis	Pointer to the instance of the VXIscInterface that is to be destroyed.

SBsessionControlGetNumChannels

The SBaudioSource and SBaudioPlayer require a Dialogic voice device for their creation routines. To create one of those resources, a SBSessionControl resource must be created for a given channel and this routine must be called to get the voice device.

```
VXIscResult SBsessionControlGetNumChannels (
 VXIunsigned    diagLogBase,
 VXIlogInterface*   log,
 VXIunsigned *  numChannels
);
```

Parameter	Description
diagLogBase	Base tag number for diagnostic logging purposes for the duration of this function call. All diagnostic tags reported start at this ID and increase upwards.
globalLog	Log used for reporting errors and diagnostic log messages for the duration of this function call.
numChannels	Pointer to the returned number of channels.

Internet interface

The OpenSpeech Browser requires a number of basic foundational components to functions including: Internet I/O, caching, and ECMAScript execution. The VXIinet, VXIcache, and VXIjsi interfaces provide these services.

The OpenSpeech Browser implements the VXIinet interface using the W3C Libwww open source Internet library. This implementation provides for document and prompt access through http:// and local files, as well as the ability to submit (POST) information via http://.

The OpenSpeech Browser implements the VXIcache interface using the W3C Libwww open source Internet library as well. This implementation provides on-disk caching of compiled grammars using a simple FIFO algorithm, where the cache persists between runs of the platform.

The OpenSpeech Browser implements the VXIjsi interface using the Mozilla SpiderMonkey open source ECMAScript (JavaScript) library. This implementation allows manipulating ECMAScript variables as well as executing arbitrary ECMAScript blocks.

The internet interfaces are:

- ❑ VXIinet — Internet Interface.
- ❑ SBinet — SBinet implementation of VXIinet.
- ❑ VXIcache — Cache Interface.
- ❑ SBcache — SBcache implementation of VXIcache.
- ❑ VXIjsi — ECMAScript (JavaScript) Engine Interface.
- ❑ SBjsi — SBjsi implementation of VXIjsi.

VXIinet

Abstract interface for accessing Internet functionality including HTTP requests, local file access, URL caching, memory buffer caching, and cookie access.

The interface is a synchronous interface based on the ANSI/ISO C standard file I/O interface, the only exception is that pre-fetches are asynchronous. The client of the interface may use this in an asynchronous manner by using non-blocking I/O operations, creating threads, or by invoking this from a separate server process.

There is one Internet interface per thread/line.

Function	Definition
VXIinet argument properties	Keys identifying properties in VXIMap for Prefetch( ), Open( ), and Unlock( ).
INET_CACHING	VXIinet caching property values.
VXIinetPrefetchPriority	INET_PREFETCH_PRIORITY property values.
INET_SUBMIT_METHOD	INET_SUBMIT_METHOD supported property values. PUT and DELETE are not supported.
VXIinet property defaults	Default values for properties in the VXIMap argument for Prefetch( ), Open( ), and Unlock( ).
VXIinetOpenMode	Mode values for Open( ).
Open flags	Flags for Open( ), may be combined through bitwise or
Open return properties	Keys identifying properties in VXIMap used to return stream information for Open( ).
INET_COOKIE	Cookie jar properties.
VXIinetResult	Result codes for interface methods.
INET platform service interface	Provides URL fetching and posting.

VXIinet argument properties

VXIinet functions take a VXIMap argument which contains a set of key/value pairs. The listed arguments must be supported by an implementation. Additional arguments can be added to this argument in other implementations. Time durations are specified in milliseconds, see below for valid values for the enumerated type properties.

Function	Definition
INET_CACHE_CONTROL_MAX_AGE	Value for the HTTP 11 Cache-Control max-age directive for requests. This specifies the client is willing to accept a cached object no older then this value (given in seconds). A value of 0 may be used to force re-validating the cached copy with the origin server for every request. In most cases, this property should not be present, thus allowing the origin server to control expiration. Value is a VXIInteger.
INET_CACHE_CONTROL_MAX_STALE	Value for the HTTP 11 Cache-Control max-stale directive for requests. This specifies the client is willing to accept a cached object that is expired by up to this value (given in seconds) past the expiration time specified by the origin server. In most cases, this property should be set to 0 or not present, thus respecting the expiration time specified by the origin server. Value is a VXIInteger.
INET_CACHING	Type of caching to apply: safe or fast. See the INET_CACHING defines. NOTE: Supported for backward compatibility only, use INET_ CACHE_CONTROL_MAX_AGE instead (“safe” mode is identical to setting that parameter to 0, while “fast” mode is identical to leaving that parameter unspecified). Value is a VXIString.
INET_OPEN_IF_MODIFIED	Conditional open, used for cases where a cache outside of VXIinet is being maintained and the desired object is already present in that cache, but the user needs to verify whether that cached object is valid for reuse or not.
INET_OPEN_LOCAL_FILE	Whether to open local files normally or to return a VXIinet_ RESULT_LOCAL_FILE error while still returning the stream information if requested. Value is a VXIInteger where it is set to TRUE (1) or FALSE (0).
INET_PREFETCH_PRIORITY	Prefetch priority. For implementations supporting priority this controls the order of opens and reads for multiple current reads. Argument is a VXIInteger.
INET_SUBMIT_METHOD	Submit method: GET or POST. Argument is a VXIString. Default is INET_SUBMIT_METHOD_DEFAULT.
INET_SUBMIT_MIME_TYPE	Submit MIME type. MIME type of data sent in the submit. Default is INET_SUBMIT_MIME_TYPE_DEFAULT. Argument is a VXIString.
INET_TIMEOUT_OPEN	Open timeout. Amount of time in (ms) to wait for an open to succeed before abandoning the open. Default is INET_TIMEOUT_ OPEN_DEFAULT Argument is a VXIInteger.

Function	Definition
INET_TIMEOUT_IO	Read timeout. Time in (ms) to attempt to read on the socket before abandoning the read. Default is INET_TIMEOUT_IO_DEFAULT. Argument is a VXIInteger.
INET_TIMEOUT_DOWNLOAD	Total download timeout. Time in (ms) to attempt to open and read all the contents on the socket before issuing a read. This is optional and should be used to implement the VoiceXML timeout. Argument is a VXIinteger.
INET_URL_BASE	URL base for resolving relative URLs. Note this is not a directory name, it is the full URL to the document that refers to this URL being fetched. No default. Argument is a VXIString.
INET_URL_QUERY_ARGS	URL Query Arguments. Argument is a VXIMap containing zero or more key/value pairs. When specified a submit is done as controlled by the INET_SUBMIT_METHOD and INET_SUBMIT_MIME_TYPE properties. For example, doing a POST where the key/value pairs are appended to the URL in order to perform a query. This is only valid for INET_MODE_READ. By default is undefined (no submit performed).

INET_CACHING

Set of defined caching property values that are used to control when to retrieve information from the cache versus when to do a fetch.

Function	Definition
INET_CACHING_SAFE	Safe caching, follows VoiceXML 1 safe fetchhint property.
INET_CACHING_FAST	Fast caching, follows VoiceXML 1 fast fetchhint property.

VXIinetPrefetchPriority

Function	Definition
INET_PREFETCH_PRIORITY_CRITICAL	Caller is waiting for the document.
INET_PREFETCH_PRIORITY_HIGH	Caller is going to get to this real soon.
INET_PREFETCH_PRIORITY_MEDIUM	Caller is likely to need this in a little bit.
INET_PREFETCH_PRIORITY_LOW	Just initializing the system, no callers yet.

INET_SUBMIT_METHOD

Full HTTP 1.1 support could be added to this interface. The current interface only supports the two methods commonly used for sending data back to a web server GET and POST.

Function	Definition
INET_SUBMIT_METHOD_GET	HTTP GET
INET_SUBMIT_METHOD_POST	HTTP POST

VXIinet property defaults

If the properties are not set in the call to these function, the given default value is assumed.

Function	Definition
INET_CACHE_CONTROL_MAX_AGE_DEFAULT	Cache Control max-age Default -NULL not present by default.
INET_CACHE_CONTROL_MAX_STALE_DEFAULT	Cache Control max-stale Default - 0, do not use expired entries.
INET_CACHING_DEFAULT	Caching Default - INET_CACHING_FAST.
INET_OPEN_IF_MODIFIED_DEFAULT	Open If Modified Default - NULL not present by default.
INET_OPEN_LOCAL_FILE_DEFAULT	Open Local File Default - TRUE.
INET_PREFETCH_PRIORITY_DEFAULT	Prefetch priority default - INET_PREFETCH_ PRIORITY_LOW.
INET_SUBMIT_METHOD_DEFAULT	Submit method default - INET_SUBMIT_METHOD_ GET.
INET_SUBMIT_MIME_TYPE_DEFAULT	Submit MIME type default - “application/x-www-form-urlencoded”.
INET_TIMEOUT_OPEN_DEFAULT	Open timeout default - 5000 (5 seconds).
INET_TIMEOUT_IO_DEFAULT	Read timeout default - 5000 (5 seconds).
INET_URL_BASE_DEFAULT	URL base default - “” (no base).
INET_URL_QUERY_ARGS_DEFAULT	URL query arguments default - NULL not present by default.

VXIinetOpenMode

Function	Definition
`INET_MODE_READ`	Open for reading, for http: access this corresponds to a GET or POST operation (GET in most cases, POST if INET_URL_QUERY_ARGS is specified and INET_ SUBMIT_METHOD is set to POST).
`INET_MODE_WRITE`	Open for writing, for http:// access this corresponds to a PUT operation.

Open flags

The Open call takes a bitwise or of open flags which control the behavior of the returned stream.

Function	Definition
INET_FLAG_NULL	Null flag. This causes the cache to use default behavior, specifically I/O using blocking operations.
INET_FLAG_NONBLOCKING_IO	Non-blocking reads/writes. Do all I/O using non-blocking operations.

Open return properties

The VXIinet implementation determines information about the file or URI when it opens the URI. These are returned as key/value pairs through a VXIMap. Some values may not return after an open.

Function	Definition
INET_INFO_ABSOLUTE_NAME	Absolute Name, always returned. The absolute URI for a URI which may have been provided as a relative URI against a base. For local file access (file:// access or an OS dependant path) an OS dependant path must be returned, never a file:// URI. This should be passed unmodified as the value of the INET_ URL_BASE property for fetching URIs referenced within this document, if any. Returned as a VXIString.
INET_INFO_MIME_TYPE	MIME type, always returned. The MIME type of the URI. For HTTP requests, this is set to the type returned by the HTTP server. For file: requests, a MIME mapping table is used to determine the correct MIME type. This table is also used when the HTTP server returns no MIME type or a generic type. If the MIME type cannot be determined at all, it is set to "application/octet-stream". Returned as a VXIString.
INET_INFO_SIZE_BYTES	Size in bytes, always returned. Size of the file in bytes.
INET_INFO_VALIDATOR	Validator, always returned on a successful Open( ) or when INET_OPEN_IF_ MODIFIED was specified and VXIinet_ RESULT_NOT_MODIFIED was returned.

INET_COOKIE

Cookie jars are represented by a VXIVector. Each element of the vector is a VXIMap that represents a cookie. The cookie VXIMap contains zero or more properties, each of which represents properties of the cookie.

The properties of the cookie VXIMap match the cookie attribute names as defined in RFC 2965. The only exceptions are as follows:

- INET_COOKIE_NAME, name of the cookie as defined in RFC 2965 - INET_COOKIE_VALUE, value of the cookie as defined in RFC 2965 - INET_COOKIE_EXPIRES, expiration time for the cookie as calculated off the MaxAge parameter defined in RFC 2965 when the cookie is accepted - RFC 2965 Discard attribute: is never returned in the VXIMap, cookies with this flag set are never returned by GetCookieJar( ).

Function	Definition
INET_COOKIE_NAME	Cookie name. Value of the key is a VXIString.
INET_COOKIE_VALUE	Cookie value key. Value of the key is a VXIString.
INET_COOKIE_EXPIRES	Cookie expires key, calculated off the Max-Age property for the cookie. Value of the key is a VXIInteger giving time since the epoch for expiration.
INET_COOKIE_COMMENT	Cookie comment, optional. Value of the key is a VXIString.
INET_COOKIE_COMMENT_URL	Cookie comment URL, optional. Value of the key is a VXIString.
INET_COOKIE_DOMAIN	Cookie domain key. Value of the key is a VXIString.
INET_COOKIE_PATH	Cookie path key. Value of the key is a VXIString.
INET_COOKIE_PORT	Cookie port key, optional. Value of the key is a VXIString.
INET_COOKIE_SECURE	Cookie secure key, optional. Value of the key is a VXIInteger set to 0 (FALSE) or 1 (TRUE).
INET_COOKIE_VERSION	Cookie standard version. Value of the key is a VXIString.

VXIinetResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Function	Definition
VXIinet_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIinet_RESULT_END_OF_STREAM	End of stream.
VXIinet_RESULT_FAILURE	Normal failure, nothing logged.
VXIinet_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIinet_RESULT_FETCH_ERROR	Other URL fetch error.
VXIinet_RESULT_FETCH_TIMEOUT	URL fetch timeout.
VXIinet_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIinet_RESULT_INVALID_PROP_NAME	Property name is not valid.
VXIinet_RESULT_INVALID_PROP_VALUE	Property value is not valid.
VXIinet_RESULT_IO_ERROR	I/O error.
VXIinet_RESULT_LOCAL_FILE	Local file, told not to open it.
VXIinet_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIinet_RESULT_NOT_FOUND	Named data not found.
VXIinet_RESULT_NOT_MODIFIED	Conditional open attempted, cached data may be used.
VXIinet_RESULT_OUT_OF_MEMORY	Out of memory.
VXIinet_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIinet_RESULT_SUCCESS	Success.
VXIinet_RESULT_SYSTEM_ERROR	System error, out of service.
VXIinet_RESULT_UNSUPPORTED	Operation is not supported.
VXIinet_RESULT_WOULD_BLOCK	I/O operation would block.

INET platform service interface

VXIinetInterface provides the URI fetch functions required by all components of the OpenSpeech browser. The ability to prefetch and read URIs along with cookie management is provided.

Function	Definition
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
Prefetch	Prefetch information (non-blocking).
Open	Open a stream for reading or writing.
Close	Close a previously opened stream.
Read	Read from a stream.
Write	Write to a stream.
SetCookieJar	Set the cookie jar.
GetCookieJar	Get the cookie jar.

Prefetch

Prefetch information (non-blocking)

```
VXIinetResult (*Prefetch)(
 struct VXIinetInterface*     pThis,
 const VXIchar  *  moduleName,
 const VXIchar*              name,
 VXIinetOpenMode    mode,
 VXIint32    flags,
 const VXIMap  *  properties
);
```

Parameter	Description
moduleName	Name of the software module that is outputting the error. See the top of VXIlog.h for moduleName allocation rules.
name	Name of the data to fetch, see Open( ) for a description of supported types.
mode	Reserved for future use, pass INET_MODE_READ.
flags	Reserved for future use, pass 0.
properties	Properties to control the prefetch, as listed above. May be NULL. Of particular note are: ❏ INET_PREFETCH_PRIORITY ❏ Hint to indicate whether and how soon to prefetch the data INET_URL_BASE ❏ Base URL for resolving relative URLs INET_URL_ QUERY_AUGS ❏ Map containing key/value pairs for HTTP queries ❏ Where the name of each key is the query argument name ❏ The value of each key is a VXIValue subclass that provides the value for that argument

Open

All implementations must support opening a URL for reading using file: or http: access, and opening a platform dependant path for reading. All implementations must support all the flags for each of the above. For all other combinations support is implementation dependant (i.e. write for URLs and platform dependant paths).

For URLs, the only accepted unsafe characters are: { } [ ] ^ ~ ' They are escaped only when processing a HTTP request.

```
VXIinetResult (*Open)(
 struct VXIinetInterface   * pThis,
 const VXIchar  *  moduleName,
 const VXIchar  *  name,
 VXIinetOpenMode    mode,
 VXIint32    flags,
 const VXIMap*    properties,
 VXIMap *   streamInfo,
 VXIinetStream  **  stream
);
```

Parameter	Description
moduleName	Name of the software module that is outputting the error. See the top of VXIlog.h for moduleName allocation rules.
name	Name of the data to fetch, See above for supported types.
mode	Mode to open the data with an INET_MODE value as defined above.
flags	Flags to control the open: INET_FLAG_NONBLOCKING_IO, non-blocking reads and writes, although the open and close is still blocking.
properties	Properties to control the open, as listed above. May be NULL. Of particular note are: ❑ INET_URL_BASE ❑ Base URL for resolving relative URLs INET_URL_ QUERY_ARGS ❑ Map containing key/value pairs for HTTP queries ❑ Where the name of each key is the query argument name ❑ The value of each key is a VXIValue subclass that provides the value for that argument.
streamInfo	(Optional, pass NULL if not required) Map that is populated with information about the stream. The INET_INFO_[...] keys listed above are mandatory with the implementation possibly setting other keys.
stream	Handle to the opened stream.

Close

Close a stream that was previously opened. Closing a NULL or previously closed stream results in an error.

```
VXIinetResult (*Close)(
 struct VXIinetInterface*     pThis,
 VXIinetStream**               stream
);
```

Parameter	Description
stream	Stream to close.

Read

This may or may not block, as determined by the flags used when opening the stream. When in non-blocking mode, partial buffers may be returned instead of blocking, or an VXIinet_RESULT_WOULD_BLOCK error is returned if no data is available at all.

```
VXIinetResult (*Read)(
 struct VXIinetInterface*     pThis,
 VXIbyte*                     buffer,
 VXIulong    buflen,
 VXIulong*                    nread,
 VXIinetStream*               stream
);
```

Parameter	Description
buffer	Buffer that receives data from the stream.
buflen	Length of buffer, in bytes.
nread	Number of bytes actual read, may be less then buflen if the end of the stream was found, or if using non-blocking I/O and there is currently no more data available to read.
stream	Handle to the stream to read from.

Write

This may or may not block, as determined by the flags used when opening the stream. When in non-blocking mode, partial writes may occur instead of blocking, or a VXIinet_RESULT_WOULD_BLOCK error is returned if no data could be written at all.

```
VXIinetResult (*Write)(
 struct VXIinetInterface*    pThis,
 const VXIbyte  *  buffer,
 VXIulong    buflen,
 VXIulong*                    nwritten,
 VXIinetStream*               stream
);
```

Parameter	Description
buffer	Buffer of data to write to the stream.
buflen	Number of bytes to write.
nread	Number of bytes actual written, may be less then buflen if an error is returned, or if using non-blocking I/O and the write operation would block.
stream	Handle to the stream to write to.

SetCookieJar

The cookie jar is used to provide cookies and store cookies during future VXIinet Prefetch ( ) and Open( ) operations. Expired cookies within the jar are not used. Each time this is called the prior cookie jar is discarded, and the caller is responsible for persistent storage of the cookie jar if desired. See GetCookieJar( ) for details.

If SetCookieJar( ) is never called, or if it is called with a NULL jar, the VXIinet implementation refuses to accept cookies for fetches.

```
VXIinetResult (*SetCookieJar)(
 struct VXIinetInterface   * pThis,
 const VXIMap  *  jar
);
```

Parameter	Description
jar	Cookie jar, specified as a VXIMap (see the description of the cookie jar properties above). Pass NULL to clear cookies on the current interface, or an empty VXIMap to provide an empty cookie jar. Pass a non-empty VXIVector as returned by GetCookieJar( ) to implement persist cookies across multiple user sessions (telephone calls).

GetCookieJar

The caller of VXIinet is responsible for persistent storage of the cookie jar if desired. This is done by calling SetCookieJar( ) with the caller's cookie jar at the start of each call (use an empty cookie jar if this is a new caller), then calling this function to retrieve the updated cookie jar at the end of the call for storage. When the cookie jar is returned, any expired cookies are deleted.

```
VXIinetResult (*GetCookieJar)(
 struct VXIinetInterface    *pThis,
 VXIMap    **jar,
 VXIbool    *changed
);
```

Parameter	Description
jar	Cookie jar, returned as a newly allocated VXIMap (see the description of the cookie jar properties above, it may be empty). The client is responsible for destroying this via VXIMapDestroy( ) when appropriate.
changed	Flag to indicate whether the cookie jar has been modified since SetCookieJar( ), allows suppressing the write of the cookie jar to persistent storage when that operation is expensive. Pass NULL if this information is not desired.

SBinet

SBinet interface, and implementation of the VXIinet abstract interface for Internet functionality including HTTP requests, local file access, URL caching, memory buffer caching, and cookie access.

Function	Definition
SBinetInit	Global platform initialization of SBinet.
SBinetShutDown	Global platform shutdown of SBinet.
SBinetCreateResource	Create a new inet service handle.
SBinetDestroyResource	Destroy the interface and free internal resources.

SBinetInit

```
VXIinetResult SBinetInit (
 VXIlogInterface   *log,
 const VXIunsigned    diagLogBase,
 const VXIchar       *reserved1,
 const int            reserved2,
 const int   reserved3,
 const int           reserved4,
 const VXIchar       *proxyServer,
 VXIulong             proxyPort,
 const VXIchar*       userAgentName,
 const VXIMap        *extensionRules,
 const VXIVector    *reserved
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging, only used for the duration of this function call.
reserved1	Reserved for future use.
reserved2	Reserved for future use.
reserved3	Reserved for future use.
reserved4	Reserved for future use.
proxyServer	Name of the proxy server to use for HTTP access, pass a server name or IP address, or NULL to do direct HTTP access.
proxyPort	Port number for accessing the proxy server.
userAgentName	HTTP user agent name sent in all HTTP messages. Must be of the form / with no spaces, such as “OSB/2.0”. When using the OpenSpeech Browser application name or a derivative, use VXI_CURRENT_VERSION_STR for the version.
extensionRules	Rules for mapping file extensions to MIME content types, used for that purpose when accessing local files and file:// URLs. Each key in the map must be an extension (period followed by the extension such as ".txt") with the value being the MIME content type for that extension. Copied internally so the pointer that is passed in still belongs to the caller.
reserved	Reserved VXIVector, pass NULL.

SBinetShutDown

```
VXIinetResult SBinetShutDown (
 VXIlogInterface   *log
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging, only used for the duration of this function call.

SBinetCreateResource

```
VXIinetResult SBinetCreateResource (
 VXIlogInterface   *log,
 const VXIchar   *cache,
 VXIinetInterface   **inet
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging, must remain a valid pointer throughout the lifetime of the resource (until SBinetDestroyResource( ) is called).
cache	VXI Cache interface used for HTTP document caching, must remain a valid pointer throughout the lifetime of the resource (until SBinetDestroyResource( ) is called)
inet	

SBinetDestroyResource

```
VXIinetResult SBinetDestroyResource (
 VXIinetInterface   **inet
);
```

Parameter	Description
inet	

VXIcache

Abstract interface for accessing caching functionality, which permits writing arbitrary data into the cache with a client supplied key name, then retrieving that data from the cache one or more times by reading against that key name.

Function	Definition
CACHE_CREATION_COST	Provides a hint on how much CPU/time went into creating the object being written cache.
CACHE_CREATION_COST_DEFAULT	Default cache creation cost.
VXIcacheOpenMode	Mode values for VXIcache::Open.
Cache Flags	Set of flags which control the caching behavior of the data put into the cache.
Cache Open Properties	Set of properties that are returned on the Open of an existing cache entry.
CACHE_OPENEX_SUPPORTED	Macros to determine the availability of new methods.
VXIcacheResult	VXIcache return codes.
VXIcacheInterface	VXIcache interface for write back caching.

CACHE_CREATION_COST

Enumerated CACHE_CREATION_COST property values. The creation costs property permits prioritizing the cache so low cost objects are flushed before high cost objects (for example, try to keep compiled grammars which have a high creation cost over keeping audio recordings that are simply fetched). Note that implementations may ignore this hint.

Function	Definition
CACHE_CREATION_COST_EXTREME	Extremely high
CACHE_CREATION_COST_HIGH	High
CACHE_CREATION_COST_MEDIUM	Medium
CACHE_CREATION_COST_LOW	Low
CACHE_CREATION_COST_FETCH	Only the cost of a fetch

CACHE_CREATION_COST_DEFAULT

Default cache creation cost.

Function	Definition
CACHE_CREATION_COST_DEFAULT	CACHE_CREATION_COST_LOW

VXIcacheOpenMode

Open supports ANSI/ISO C standard I/O open modes which can be bitwise or together to get the open mode.

Function	Definition
CACHE_MODE_READ	Open for reading
CACHE_MODE_WRITE	Open for writing

Cache Flags

The Open call takes bitwise or cache flags that control the caching behavior of data which is written or read from the cache.

Function	Definition
CACHE_FLAG_NULL	Null flag
CACHE_FLAG_LOCK	Never flush from the cache
CACHE_FLAG_LOCK_MEMORY	Never flush from memory
CACHE_FLAG_NONBLOCKING_IO	Non-blocking reads/writes

Cache open properties

The Open call returns a VXIMap which contains a set of key/value properties. The following provides the definition of the properties that must be supported by an implementation of Open.

Function	Definition
CACHE_INFO_FINAL_KEY	Final key name used for storing the cache entry.
CACHE_INFO_LAST_MODIFIED	Last Modified time.
CACHE_INFO_SIZE_BYTES	Size in bytes.

VXIcacheResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Function	Definition
VXIcache_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIcache_RESULT_IO_ERROR	I/O error.
VXIcache_RESULT_OUT_OF_MEMORY	Out of memory.
VXIcache_RESULT_SYSTEM_ERROR	System error, out of service.
VXIcache_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIcache_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIcache_RESULT_INVALID_PROP_NAME	Property name is not valid.
VXIcache_RESULT_INVALID_PROP_VALUE	Property value is not valid.
VXIcache_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIcache_RESULT_SUCCESS	Success.
VXIcache_RESULT_FAILURE	Normal failure, nothing logged.
VXIcache_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIcache_RESULT_NOT_FOUND	Named data not found.
VXIcache_RESULT_NULL_STREAM	I/O operation handed a NULL stream.
VXIcache_RESULT_WOULD_BLOCK	I/O operation would block.
VXIcache_RESULT_END_OF_STREAM	End of stream.
VXIcache_RESULT_EXCEED_MAXSIZE	Buffer exceeds maximum size.
VXIcache_RESULT_ENTRY_LOCK	Entry in the cache in currently in used and is locked.
VXIcache_RESULT_UNSUPPORTED	Operation is not supported.

VXIcacheInterface

VXIcacheInterface provides I/O functions for reading and writing items into the cache. In addition, cache entries that where locked by the open can be unlocked using Unlock.

Function	Definition
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
Open	Open a stream for reading or writing.
Close	Close a previously opened stream.
Unlock	Unlock a stream that was previously locked into the cache.
Read	Read from a stream.
Write	Write to a stream.

SBcache

SBcache interface, and implementation of the VXIcache abstract interface which permits writing arbitrary data into the cache with a client supplied key name, then retrieving that data from the cache one or more times by reading against that key name.

Function	Definition
SBcacheInit	Global platform initialization of SBcache.
SBicacheShutDown	Global platform shutdown of SBcache.
SBcacheCreateResource	Create a new cache service handle.
SBcacheDestroyResource	Destroy the interface and free internal resources.

SBcacheInit

```
VXIcacheResult SBcacheInit (
 VXIlogInterface    *log,
 const VXIunsigned    diagLogBase,
 const VXIchar   *cacheDir,
 const int   cacheSizeMB,
 const int   entryMaxSizeMB,
 const int   entryExpTimeSec,
 VXIbool   unlockEntries
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.
diagLogBase	Base diagnostic logging tag ID used for all diagnostic logging message. (See SBprompt and all the other descriptions of diagLogBase.)
cacheDir	Cache directory name.
cacheSizeMB	Maximum size of the data in the cache directory, in megabytes.
entryMaxSizeMB	Maximum size of any individual cache entry, in megabytes
entryExpTimeSec	Maximum amount of time any individual cache entry remains in the cache, in seconds.
unlockEntries	TRUE to unlock locked entries on startup (from using CACHE_FLAG_LOCK and CACHE_FLAG_LOCK_ MEMORY) FALSE to leave them locked.

SBcacheShutDown

```
VXIcacheResult SBcacheShutDown (
 VXIlogInterface    *log
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.

SBcacheCreateResource

```
VXIcacheResult SBcacheCreateResource (
 VXIlogInterface   *log,
 VXIcacheInterface   **cache
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging, must remain a valid pointer throughout the lifetime of the resource (until SBcacheDestroyResource is called).

SBcacheDestroyResource

```
VXIcacheResult SBcacheDestroyResource (
 VXIcacheInterface   **cache
);
```

Parameter	Description
cache	Once this is called, the logging interface passed to SBcacheCreateResource( ) may be released.

VXIjsi

Abstract interface for interacting with a JavaScript (ECMAScript) engine. This provides functionality for:

- creating ECMAScript execution contexts
- manipulating ECMAScript scopes
- manipulating variables within those scopes
- evaluating ECMAScript expressions/scripts

Function	Definition
VXIjsiResult	Result codes for interface methods.
VXIjsiInterface	VXIjsi interface for ECMAScript evaluation.

VXIjsiResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Function	Definition
VXIjsi_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIjsi_RESULT_IO_ERROR	I/O error.
VXIjsi_RESULT_OUT_OF_MEMORY	Out of memory.
VXIjsi_RESULT_SYSTEM_ERROR	System error, out of service.
VXIjsi_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIjsi_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIjsi_RESULT_INVALID_PROP_NAME	Property name is not valid.
VXIjsi_RESULT_INVALID_PROP_VALUE	Property value is not valid.
VXIjsi_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIjsi_RESULT_SUCCESS	Success.
VXIjsi_RESULT_FAILURE	Normal failure, nothing logged.
VXIjsi_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIjsi_RESULT_SYNTAX_ERROR	JavaScript syntax error.
VXIjsi_RESULT_SCRIPT_EXCEPTION	JavaScript exception thrown.
VXIjsi_RESULT_SECURITY_VIOLATION	JavaScript security violation.
VXIjsi_RESULT_LOCAL_FILE	Local file, told not to open it.

Function	Definition
VXIjsi_RESULT_NULL_STREAM	I/O operation handed a NULL stream.
VXIjsi_RESULT_NOT_MODIFIED52	Conditional open attempted, cached data may be used.
VXIjsi_RESULT_UNSUPPORTED	Operation is not supported.

VXIjsiInterface

Function	Definition
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
Create Context	Create and initialize a new script context.
DestroyContext	Destroy a script context, clean up storage if required.
CreateVarExpr	Queries the Create a script variable relative to the current scope, initialized to an expression of the line.
CreateVarValue	Create a script variable relative to the current scope, initialized to a VXIValue based value.
SetVarExpr	Set a script variable to an expression relative to the current scope.
SetVarValue	Set a script variable to a value relative to the current scope.
GetVar	Get the value of a variable.
CheckVar	Check whether a variable is defined (not ECMAScript Undefined).
Eval	Execute a script, optionally returning any execution result.
PushScope	Push a new context onto the scope chain (add a nested scope).
PopScope	Pop a context from the scope chain (remove a nested scope).
ClearScopes	Reset the scope chain to the global scope (pop all nested scopes).

CreateContext

This creates a new context. Currently one context is created per thread, but the implementation must support the ability to have multiple contexts per thread.

```
VXIjsiResult (*CreateContext)(
 struct VXIjsiInterface   * pThis,
 VXIjsiContext   **  context
);
```

Parameter	Description
context	Newly created context.

DestroyContext

```
VXIjsiResult (*DestroyContext)(
 struct VXIjsiInterface*    pThis,
 VXIjsiContext**    context
);
```

Parameter	Description
context	Context to destroy.

CreateVarExpr

When there is an expression, it gets evaluated, then the value of the evaluated expression (the final sub-expression) assigned. Thus an expression of “1; 2;” actually assigns 2 to the variable.

```
VXIjsiResult (*CreateVarExpr)(
 struct VXIjsiInterface*     pThis,
 VXIjsiContext*    context,
 const VXIchar*              name,
 const VXIchar*              expr
);
```

Parameter	Description
context	ECMAScript context to create the variable within.
name	Name of the variable to create.
expr	Expression to set the initial value of the variable (if NULL or empty the variable is set to ECMAScript Undefined as required for VoiceXML).

CreateVarValue

```
VXIjsiResult (*CreateVarValue)(
 struct VXIjsiInterface* pThis,
 VXIjsiContext*    context,
 const VXIchar*              name,
 const VXIchar*              expr
);
```

Parameter	Description
context	ECMAScript context to create the variable within.
name	Name of the variable to create.
value	VXIValue based value to set the initial value of the variable (if NULL the variable is set to ECMAScript Undefined as required for VoiceXML). VXIMap is used to pass ECMAScript objects.

SetVarExpr

The expression is evaluated, then the value of the evaluated expression (the final sub-expression) assigned. Thus an expression of "1; 2;" actually assigns 2 to the variable.

```
VXIjsiResult (*SetVarExpr)(
 struct VXIjsiInterface* pThis,
 VXIjsiContext*    context,
 const VXIchar*              name,
 const VXIchar*              expr
);
```

Parameter	Description
context	ECMAScript context to set the variable within.
name	Name of the variable to set.
expr	Expression to be assigned.

SetVarValue

```
VXIjsiResult (*SetVarValue)(
 struct VXIjsiInterface* pThis,
 VXIjsiContext*    context,
 const VXIchar*              name,
 const VXIchar*              expr
);
```

Parameter	Description
context	ECMAScript context to set the variable within.
name	Name of the variable to set.
value	VXIValue based value to be assigned. VXIMap is used to pass ECMAScript objects.

GetVar

```
VXIjsiResult (*GetVar)(
 struct VXIjsiInterface*    pThis,
 const VXIjsiContext*    context,
 const VXIchar*          name,
 VXIValue **   value
);
```

Parameter	Description
context	ECMAScript context to get the variable from.
name	Name of the variable to get.
value	Value of the variable, which returns as the VXI type that most closely matches the variable's ECMAScript type. This function allocates this for return on success (returns a NULL pointer otherwise), and the caller is responsible for destroying it via VXIValueDestroy( ). VXIMap is used to return ECMAScript objects.

CheckVar

A variable with a ECMAScript Null value is considered defined.

```
VXIjsiResult (*CheckVar)(
 struct VXIjsiInterface* pThis,
 const VXIjsiContext*    context,
 const VXIchar*          name
);
```

Parameter	Description
context	ECMAScript context to check the variable in.
name	Name of the variable to check.

Eval

```
VXIjsiResult (*Eval)(
 struct VXIjsiInterface* pThis,
 VXIjsiContext*          context,
 const VXIchar*          expr,
 VXIValue **   result
);
```

Parameter	Description
context	ECMAScript context to execute within.
expr	Buffer containing the script text.
value	Result of the script execution, which returns as the VXI type that most closely matches the variable’s ECMAScript type. Pass NULL if the result is not desired. Otherwise the function allocates this for return on success when there is a return value (returns a NULL pointer otherwise), and the caller is responsible for destroying it via VXIValueDestroy( ). VXIMap is used to return JavaScript objects.

PushScope

```
VXIjsiResult (*PushScope)(
 struct VXIjsiInterface* pThis,
 VXIjsiContext*          context,
 const VXIchar*          name
);
```

Parameter	Description
context	ECMAScript context to push the scope onto.
name	Name of the scope. This is used to permit referencing variables from an explicit scope within the scope chain, such as “myscope.myvar” to access “myvar” within a scope named “myscope”.

PopScope

```
VXIjsiResult (*PopScope)(
 struct VXIjsiInterface* pThis,
 VXIjsiContext*          context
);
```

Parameter	Description
context	ECMAScript context to pop the scope from.

ClearScopes

```
VXIjsiResult (*ClearScopes)(
 struct VXIjsiInterface* pThis,
 VXIjsiContext*          context
);
```

Parameter	Description
context	ECMAScript context to clear the scope from.

SBjsi

SBjsi interface, an implementation of the VXIjsi interface for interacting with a ECMAScript (JavaScript) engine. This provides functionality for creating ECMAScript execution contexts, manipulating ECMAScript scopes, manipulating variables within those scopes, and evaluating ECMAScript expressions/scripts.

Function	Definition
SBjsiInit	Global platform initialization of JavaScript.
SBjsiShutDown	Global platform shutdown of JavaScript.
SBjsiCreateResource	Create a new JavaScript service handle.
SBjsiDestroyResource	Destroy the interface and free internal resources.

SBjsiInit

```
VXIjsiResult SBjsiInit (
 VXIlogInterface   *log,
 VXIunsigned   diagLogBase,
 VXIlong   runtimeSize,
 VXIlong   contextSize,
 VXIlong   maxBranches
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.
diagLogBase	Base tag number for diagnostic logging purposes. All diagnostic tags for SBjsi starts at this ID and increase upwards.
runtimeSize	Size of the JavaScript runtime environment, in bytes. There is one runtime per process.
contextSize	Size of each JavaScript context, in bytes. There may be multiple contexts per channel, although the VXI typically only uses one per channel.
maxBranches	Maximum number of JavaScript branches for each JavaScript evaluation, used to interrupt infinite loop from (possibly malicious) scripts.

SBjsiShutDown

```
VXIjsiResult SBjsiShutDown (
 VXIlogInterface  * log
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.

SBjsiCreateResource

```
VXIjsiResult SBjsiCreateResource(
 VXIlogInterface  * log,
 VXIjsiInterface  ** jsi
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging, must remain a valid pointer throughout the lifetime of the resource (until SBjsiDestroyResource( ) is called).
jsi	Returned resource interface.

SBjsiDestroyResource

```
VXIjsiResult SBjsiDestroyResource(
 VXIjsiInterface    **jsi
);
```

Parameter	Description
jsi	Once this is called, the logging interface passed to SBjsiCreateResource( ) may be released.

Logging interface

The OpenSpeech Browser logs through the VXIlog interface, which provides a mechanism for error logging, event logging, and diagnostic logging.

The OpenSpeech Browser includes a logging implementation called SBlog that processes logging requests and forwards the results to one or more log listeners (C callbacks) for output.

Reference log listeners are provided in source form that output errors and diagnostic messages to a file and optionally the console, with event messages being output to the OpenSpeech Recognizer event log file. See src/HW/client/api/SBlogListeners.cpp in the distribution for these reference listeners, which are built into SBclient.dll.

The logging interfaces are:

- VXIlog — Abstract interface for logging errors, diagnostic logging, and event logging.
- SBlog — SBlog implementation of VXIlog.
- SBlogListeners — Reference implementation of SBlog listeners.

VXIlog

Abstract interface for logging errors, diagnostic logging, and event logging.

Function	Definition
VXIlogEvent	Standard VXIlog events.
VXIlogResult	Result codes for interface methods.
VXIlogInterface	Defines the Logging service interface.

The VXIlog Facility supports the following types of log streams:

- Error stream

  Used to report system faults (errors) and possible system faults (warnings) to the system operator for diagnosis and repair. Errors are reported by error numbers that are mapped to text with an associated severity level. This mapping may be maintained in a separate XML document. This allows integrators to localize and customize the text without code changes, and also allows integrators and developers to review and re-define severity levels without code changes. Optional attribute/value pairs are used to convey details about the fault, such the file or URL that is associated with the fault.

- Diagnostic stream

  Used by developers and support staff to trace and diagnose system behavior. Diagnostic messages are hard-coded text since they are private messages that are not intended for system operator use. Diagnostic messages are all associated with developer defined “tags” for grouping, where each tag may be independently enabled/disabled. Diagnostic logging has little or no performance impact if the tag is disabled, so developers should be generous in their insertion of diagnostic log messages. These are disabled by default, but specific tags can then be enabled for rapid diagnosis with minimal performance impact when issues arise.

- Event stream

  Summarizes normal caller activity in order to support service bureau billing; reporting for capacity planning, application and recognition monitoring, and caller behavior trending; and traces of individual calls for application tuning. Events are reported by event numbers that are mapped to event names. Optional attribute/value pairs are used to convey details about the event, such as the base application for a new call event.

❑ Content stream

The content stream is not a separate conceptual stream: it is a mechanism to log large blocks of data (BLOBs) in a flexible and efficient manner. For example, diagnostic logging of the VoiceXML page being executed, or event logging of the audio input for recognition. To log this information, a request to open a content stream handle is made, and then the data is written to that stream handle. Next the client uses a key/value pair returned by the VXIlog implementation to cross-reference the content within an appropriate error, diagnostic, or event log entry. This mechanism gives the VXIlog implementation high flexibility: the implementation may simply write each request to a new file and return a cross-reference key/value pair that indicates the file path (key = "URL", value =), or may POST the data to a centralized web server, or write each request to a database. The system operator console software may then transparently join the error, diagnostic, and event log streams with the appropriate content data based on the key/value pair.

Across all streams, the log implementation is responsible for automatically supplying the following information in some manner (possibly encoded into a file name, possibly in a data header, possibly as part of each log message) for end consumer use:

- timestamp
- channel name and/or number
- host name
- application name
- (Error only) error text and severity based on the error number, and the supplied module name
- (Diagnostic only) tag name based on the tag number, and the supplied subtag name
- (Event only) event name based on the event number

In addition, for diagnostic logging the log implementation is responsible for defining a mechanism for enabling/disabling messages on an individual tag basis without requiring a recompile for use by consumers of the diagnostic log. It is critical that Diagnostic( ) is highly efficient for cases when the tag is disabled: in other words, the lookup for seeing if a tag (an integer) is enabled should be done using a simple array or some other extremely low-overhead mechanism. It is highly recommended that log implementations provide a way to enable/disable tags on the fly (without needing to restart the software), and log implementations should consider providing a way to enable/disable tabs on a per-channel basis (for enabling tags in tight loops where the performance impact can be large).

Each of the streams has fields that need to be allocated by developers. The rules for each follow. As background, several of these fields require following the rules for XML names: it must begin with a letter, underscore, or colon, and is followed by one or more of those plus digits, hyphens, or periods. However, colons must only be used when indicating XML namespaces, and the "vxi" and "swi" namespaces (such as "swi:SBprompt") are reserved for use by SpeechWorks International, Inc.

❑ Error logging

Module names (moduleName) must follow XML name rules as described above, and must be unique for each implementation of each VXI interface.

Error IDs (errorID) are unsigned integers that for each module start at 0 and increase from there, allocated by each named module as it sees fit. Each VXI interface implementation must provide a document that provides the list of error IDs and the recommended text (for at least one language) and severity. Severities should be constrained to one of three levels:

- “Critical - out of service” (affects multiple callers)
- “Severe - service affecting” (affects a single caller)
- “Warning - not service affecting” (not readily apparent to callers, or at the very least, callers do not notice)

Attribute names must follow XML name rules as described above. However, these need not be unique for each implementation as they are interpreted relative to the module name (and frequently not interpreted at all, but merely output). For consumer convenience the data type for each attribute name should never vary, although most log implementations do not enforce this.

❑ Diagnostic logging

Tags (tagID) are unsigned integers that must be globally unique across the entire run-time environment to allow VXIlog implementations to have very low overhead diagnostic log enablement lookups (see above). The recommended mechanism for avoiding conflicts with components produced by other developers is to make it so the initialization function for your component takes an unsigned integer argument that is a tag ID base. Then within your component code, allocate locally unique tag ID numbers starting at zero, but offset them by the base variable prior to calling the VXIlog Diagnostic( ) method. This way integrators of your component can assign a non-conflicting base as they see fit.

There are no restrictions on subtag use, as they are relative to the tag ID and most log implementations merely output them as a prefix to the diagnostic text.

❑ Event logging

Events (eventID) are unsigned integers that are defined by each developer as they see fit in coordination with other component developers to avoid overlaps. Globally unique events are required to make it easy for event log consumers to parse the log; all events should be well known and well documented.

Attribute names must follow XML name rules as described above. However, these need not be unique for each implementation as they are interpreted relative to the module name (and frequently not interpreted at all, but merely output). For consumer convenience the data type for each attribute name should never vary, although most log implementations do not enforce this.

- Content logging

  Module names (moduleName) must follow XML name rules as described above, and must be unique for each implementation of each VXI interface.

  When a content stream is opened for writing data, a key/value pair is returned by the VXIlog interface. This key/value pair must be used to cross-reference this data in error, diagnostic, and/or event logging messages.

VXIlogEvent

Standardized events that may be reported to the VXIlog interface. Platform dependant events start at VXIlog_EVENT_PLATFORM_DEFINED and increase from there.

LOG_CONTENT_METHODS_SUPPORTED

Macros to determine the availability of new methods.

VXIlogResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues)

Feature	Description
VXIlog_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIlog_RESULT_IO_ERROR	I/O error.
VXIlog_RESULT_OUT_OF_MEMORY	Out of memory.
VXIlog_RESULT_SYSTEM_ERROR	System error, out of service.
VXIlog_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIlog_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIlog_RESULT_INVALID_PROP_NAME	Property name is not valid.
VXIlog_RESULT_INVALID_PROP_VALUE	Property value is not valid.
VXIlog_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIlog_RESULT_SUCCESS	Success.
VXIlog_RESULT_FAILURE	Normal failure, nothing logged.
VXIlog_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIlog_RESULT_UNSUPPORTED	Operation is not supported.

VXIlogInterface

The VXIlogInterface provides a set of functions which are used for logging by all components OpenSpeech Browser components.

Feature	Description
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
Error	Log an error.
VError	Log an error (va_list variant).
Diagnostic	Log a diagnostic message.
VDiagnostic	Log a diagnostic message (va_list variant).
DiagnosticIsEnabled	Query whether diagnostic logging is enabled.
Event	Log an event.
VEvent	Log an event (va_list variant).
ContentOpen	Open a handle to log (potentially large or binary) content. NOTE: This is only available in revision 1.1 of the VXIlogInterface, use LOG_CONTENT_METHODS_SUPPORTED( ) to determine availability.
ContentClose	Close a stream for logging (potentially large or binary) content. NOTE: This is only available in revision 1.1 of the VXIlogInterface, use LOG_CONTENT_METHODS_SUPPORTED( ) to determine availability.
ContentWrite	Write (potentially large or binary) content to a logging stream. NOTE: This is only available in revision 1.1 of the VXIlogInterface, use LOG_CONTENT_METHODS_SUPPORTED( ) to determine availability.

Error

Basic error reporting mechanism. Errors are reported by moduleName, error number, a format, and a varargs argument list.

IMPORTANT: Error details are not free form, they must be passed as a succession of key-value pairs, i.e. a string key followed by a value. For example, this format string and arguments is correct:

```
L"%s%i%s%s", L"key1", 911, L"key2", L"value2"
```

While this one is incorrect (second key missing):

```
L"%s%i%f", L"key1", 911, (float)22 / 7
```

Keys must always be specified by a %s, and the key names must follow the rules for XML names as summarized at the top of this header. Values may be specified by the ANSI C defined format parameters for printf( ), including the ability to control string widths, number of decimals to output, padding etc. There are no restrictions on the variable values, it is the responsibility of the logging system to escape the variable values if required by the final output stream (such as output via XML).

NOTE

Do NOT use %C and %S in the format string for inserting narrow character buffers (char and char *) as supported by some compilers, as this is not portable and may result in system crashes on some UNIX variants if the VXIlog implementation uses the compiler supplied printf( ) family of functions for handling these variable argument lists.

```
VXIlogResult (*Error)(
 struct VXIlogInterface*     pThis,
 const VXIchar*     moduleName,
 VXIunsigned     errorID,
 const VXIchar*     format,
       ...
);
```

Parameter	Description
moduleName	Name of the software module that is outputting the error. See the top of this file for moduleName allocation rules.
errorID	Error number to log, this is mapped to localized error text that is displayed to the system operator that has an associated severity level. It is CRITICAL that this provides primary, specific, actionable information to the system operator, with attribute/value pairs only used to provide details. See the top of this file for errorID allocation rules.
format	Format string as passed to wprintf (the wchar_t version of printf( ) as defined by the ANSI C standard) for outputting optional error details. This is followed by a variable list of arguments supplying variables for insertion into the format string, also as passed to wprintf( ).
...	Arguments matching the error details format specified above.

VError

Same as Error, but a va_list is supplied as an argument instead of "..." in order to make it easy to layer convenience functions/classes for logging on top of this interface.

```
VXIlogResult (*VError)(
 struct VXIlogInterface*    pThis,
 const VXIchar*    moduleName,
 VXIunsigned                errorID,
 const VXIchar*              format,
 va_list                     vargs
);
```

Parameter	Description
moduleName	Name of the software module that is outputting the error. See the top of this file for moduleName allocation rules.
errorID	Error number to log, this is mapped to localized error text that is displayed to the system operator that has an associated severity level. It is CRITICAL that this provides primary, specific, actionable information to the system operator, with attribute/value pairs only used to provide details. See the top of this file for errorID allocation rules.
format	Format string as passed to wprintf (the wchar_t version of printf( ) as defined by the ANSI C standard) for outputting optional error details. This is followed by a variable list of arguments supplying variables for insertion into the format string, also as passed to wprintf( ).
vargs	Arguments matching the error details format specified above.

Diagnostic

Basic diagnostic reporting mechanism. Diagnostic messages are reported by moduleName, tag id, subtag, a format, and a variable length argument list.

```
VXIlogResult (*Diagnostic)(
 struct VXIlogInterface* pThis,
 VXIunsigned             tagID,
 const VXIchar*          subtag,
 const VXIchar*          format,
     ...
);
```

Parameter	Description
tagID	Identifier that classifies a group of logically associated diagnostic messages (usually from a single software module) that are desirable to enable or disable as a single unit.
subtag	Arbitrary string that may be used to subdivide the diagnostic messages of that tagID, or provide additional standardized information such as the source file, function, or method. There are no rules for the content of this field.
format	Format string as passed to wprintf( ) (the wchar_t version of printf( ) as defined by the ANSI C standard) for outputting free-form diagnostic text. This is followed by a variable list of arguments that supply values for insertion into the format string, also as passed to wprintf( ). Do NOT use %C and %S in the format string for inserting narrow character strings (char and char *) as supported by some compilers, as this is not portable and may result in system crashes on some UNIX variants if the VXIlog implementation uses the compiler supplied ...printf( ) family of functions for handling these variable argument lists.
...	Arguments matching the free-form diagnostic text format specified above.

VDiagnostic

Same as Diagnostic, but a va_list is supplied as an argument instead of "..." in order to make it easy to layer convenience functions/classes for logging on top of this interface.

```
VXIlogResult (*VDiagnostic)(
 struct VXIlogInterface* pThis,
 VXIunsigned             tagID,
 const VXIchar*          subtag,
 const VXIchar*          format,
 va_list                 vargs
);
```

Parameter	Description
tagID	Identifier that classifies a group of logically associated diagnostic messages (usually from a single software module) that are desirable to enable or disable as a single unit.
subtag	Arbitrary string that may be used to subdivide the diagnostic messages of that tagID, or provide additional standardized information such as the source file, function, or method. There are no rules for the content of this field.
format	Format string as passed to wprintf( ) (the wchar_t version of printf( ) as defined by the ANSI C standard) for outputting free-form diagnostic text. This is followed by a variable list of arguments that supply values for insertion into the format string, also as passed to wprintf( ). Do NOT use %C and %S in the format string for inserting narrow character strings (char and char *) as supported by some compilers, as this is not portable and may result in system crashes on some UNIX variants if the VXIlog implementation uses the compiler supplied ...printf( ) family of functions for handling these variable argument lists.
vargs	Arguments matching the free-form diagnostic text format specified above.

DiagnosticIsEnabled

Diagnostic log messages are automatically filtered in a high-performance way by the diagnostic method. This should only be used in the rare conditions when there is significant pre-processing work required to assemble the input parameters for Diagnostic( ), and thus it is best to suppress that performance impacting pre-processing as well.

```
VXIbool (*DiagnosticIsEnabled)(
 struct VXIlogInterface* pThis
 VXIunsigned             tagID
);
```

Parameter	Description
DiagnosticIsEnabled	Query whether diagnostic logging is enabled.
tagID	Identifier for a class of

Event

Basic error reporting mechanism. Errors are reported by moduleName, error number, and with a varargs format. Event details are not free form, they must be passed as a succession of key-value pairs, i.e. a string key followed by a value. See the description of the format parameter for Error for a full explanation.

```
VXIlogResult (*Event)(
 struct VXIlogInterface* pThis,
 VXIunsigned             eventID,
 const VXIchar*          format,
     ...
);
```

Parameter	Description
eventID	Event number to log, this is mapped to a localized event name that is placed in the event log. It is critical that this provide unambiguous information about the nature of the event. See the top of this file for eventID allocation rules.
eventHandle	Returned handle to the event, used to add attribute/value pairs that provide details about the event (such as the grammar name, etc.).
format	Format string as passed to wprintf( ) (the wchar_t version of printf( ) as defined by the ANSI C standard) for outputting optional event details. This is followed by a variable list of arguments supplying values for insertion into the format string, also as passed to wprintf(). IMPORTANT: Event details are not free form, they must be passed as a succession of key-value pairs, i.e. a string key followed by a value. See the description of the format parameter for Error for a full explanation.
...	Arguments matching the event details format specified above.

VEvent

Same as Event, but a va_list is supplied as an argument instead of "..." in order to make it easy to layer convenience functions/classes for logging on top of this interface.

```
VXIlogResult (*VEvent)(
 struct VXIlogInterface* pThis,
 VXIunsigned             eventID,
 const VXIchar*          format,
 va_list                 vargs
);
```

Parameter	Description
eventID	Event number to log, this is mapped to a localized event name that is placed in the event log. It is critical that this provide unambiguous information about the nature of the event. See the top of this file for eventID allocation rules.
eventHandle	Returned handle to the event, used to add attribute/value pairs that provide details about the event (such as the grammar name, etc.).
format	Format string as passed to wprintf( ) (the wchar_t version of printf( ) as defined by the ANSI C standard) for outputting optional event details. This is followed by a variable list of arguments supplying values for insertion into the format string, also as passed to wprintf(). IMPORTANT: Event details are not free form, they must be passed as a succession of key-value pairs, i.e. a string key followed by a value. See the description of the format parameter for Error for a full explanation.
vargs	Arguments matching the event details format specified above.

ContentOpen

This method is used to open a content logging stream in situations where large blocks of data need to be logged and/or the data is binary. Data is written via ContentWrite( ), and the stream is then closed via ContentClose( ). The key/value pair returned by this method indicates the location of the logged data, and should be used to reference this content within error, event, and/or diagnostic messages.

```
VXIlogResult (ContentOpen)(
 const VXIchar*    moduleName,
 const VXIchar*          contentType,
 VXIstring**              logKey,
 VXIstring**              logValue,
 VXIlogStream**    stream
);
```

Parameter	Description
moduleName	Name of the software module that is outputting the data. See the top of this file for moduleName allocation rules.
contentType	Content type for the data.
key	Key name to cross-reference this content in logging errors, events, and/or diagnostic messages. Ownership is passed on success, call VXIStringDestroy( ) to free this when no longer required.
value	Value name to cross-reference this content in logging errors, events, and/or diagnostic messages. Ownership is passed on success, call VXIStringDestroy( ) to free this when no longer required.
stream	Handle for writing the content via ContentWrite( ) and closing it via ContentClose( ).

ContentClose

Close a content stream that was previously opened. Closing a NULL or previously closed stream results in an error.

```
VXIlogResult (ContentClose)(
 VXIlogStream**    stream
);
```

Parameter	Description
stream	Handle to the stream to close, is set to NULL on success.

ContentWrite

Write data to a content stream that was previously opened.

```
VXIlogResult (ContentWrite)(
 struct VXIlogInterface*    pThis,
 const VXIbyte*    buffer,
 VXIulong              buflen,
 VXIulong*              nwritten,
 VXIlogStream**    stream
);
```

Parameter	Description
buffer	Buffer of data to write to the stream.
buflen	Number of bytes to write.
nwritten	Number of bytes actual written, may be less then buflen if an error is returned.
stream	Handle to the stream to write to.

SBlog

SBlog is an implementation of the VXIlogInterface for logging. The SBlogInterface extends the VXIlogInterface to support a listener registration mechanism for sending the logs to the final output stream. This allows multiple systems to tap on to a single log event which can be sent to a file and a central logging service if desired. Listeners are invoked through callbacks. The callback call is blocking. If a listener is going to do significant work or invoke functions which may block, for example a socket write, a message queue should be implemented so that callback is non-blocking. Failure to do this impacts overall system performance.

SBlog is responsible for automatically supplying the following information for end consumer use across all streams,:

- timestamp
- (Error only) error key/values, the error number, and the supplied module name.
- (Diagnostic only) tag number, and the supplied subtag name
- (Event only) key/values pairs to log in the event

In addition, for diagnostic logging the SBlog defines a mechanism for enabling/disabling messages on an individual tag basis without requiring a recompile for use by consumers of the diagnostic log. Diagnostic( ) is efficient for cases when the tag is disabled: in other words, the lookup for seeing if a tag (an integer) is enabled should be done using a simple array or other extremely low-overhead mechanisms. SBlog provides a way to enable/disable tags on the fly (without needing to restart the software).

Each OSB PIK component provides an XML error file which integrators can choose to use (or rewrite) to map error numbers and error key/values to error text. An XSLT transform can do this in an error viewer.

Function	Definition
SBlogStream	SBlog definition of a VXIlogStream.
SBlogErrorListener	Prototype for error listener notification.
SBlogDiagnosticListener	Prototype for diagnostic listener notification.
SBlogEventListener	Prototype for event listener notification.
SBlogContentListener	Prototype for content listener notification.
SBlogInterface	SBlog extension interface to the VXIlog interface.
SBlogInit	Global platform initialization of SBlog.
SBlogShutDown	Global platform shutdown of SBlog.
SBlogCreateResource	Create a new log service handle.
SBlogDestroyResource	Destroy the interface and free internal resources.

SBLogStream

SBlog defines a log stream as the following, which contains methods for writing data and closing the stream.

Function	Definition
Close	Close the stream.
Write	Write content to the stream.

SBlogErrorListener

All error listener registrants must conform to this signature.

```
SBlogErrorListener(
 SBlogInterface*    pThis,
 const VXIchar*    moduleName,
 VXIunsigned    errorID,
 time_t    timestamp,
 VXIunsigned    timestampMsec,
 const VXIVector*    keys,
 const VXIVector*    values,
 void*   userdata
);
```

Parameter	Description
pThis	Pointer to the SBlogInterface that issued the callback.
moduleName	Module name on which the error occurred.
errorID	Error number.
timestamp	time_t for the time of the log event.
timestampMsec	Milliseconds for the time of the log event.
keys	VXIVector of keys, all VXIString types.
values	VXIVector of values, each a VXIInteger, VXIFloat, or VXIPtr.
userdata	User data that is delivered in the callback.

SBlogDiagnosticListener

All diagnostic listener registrants must conform to this signature.

```
SBlogDiagnosticListener(
 SBlogInterface  * pThis,
 VXIunsigned   tagID,
 const VXIchar*   subtag,
 time_t    timestamp,
 VXIunsigned   timestampMsec,
 const VXIchar*   printmsg,
 void*  userdata
);
```

Parameter	Description
pThis	Pointer to the SBlogInterface that issued the callback.
tagID	Identifier that classifies a group of logically associated diagnostic messages (usually from a single software module) that are desirable to enable or disable as a single unit.
subtag	Arbitrary string that may be used to subdivide the diagnostic messages of that tagID, or provide additional standardized information such as the source file, function, or method. There are no rules for the content of this field.
timestamp	time_t the time of the log event.
timestampMsec	Milliseconds for the time of the log event.
printmsg	NULL terminated string to be printed.
userdata	User data that is delivered in the callback.

SBlogEventListener

All event listener registrants must conform to this signature.

```
SBlogEventListener(
 SBlogInterface*   pThis,
 VXIunsigned   eventID,
 time_t   timestamp,
 VXIunsigned   timestampMsec,
 const VXIVector*  keys,
 const VXIVector*  values,
 void*  userdata
);
```

Parameter	Description
pThis	Pointer to the SBlogInterface that issued the callback.
errorID	Event number.
timestamp	time_t for the time of the log event.
timestampMsec	Milliseconds for the time of the log event.
keys	VXIVector of keys, all VXIString types.
values	VXIVector of values, each a VXIInteger, VXIFloat, or VXIPtr.
userdata	User data that is delivered in the callback.

SBlogContentListener

All content listener registrants must conform to this signature.

```
SBlogContentListener(
 SBlogInterface*   pThis,
 const VXIchar*   moduleName,
 const VXIchar*    contentType,
 void*    userdata,
 VXIString**    logKey,
 VXIString**    logValue,
 SBlogStream**    stream
);
```

Parameter	Description
pThis	Pointer to the SBlogInterface that issued the callback.
moduleName	Name of the software module that is outputting the data.
contentType	MIME content type for the data.
userdata	User data that is delivered in the callback.
Key	Key name to cross-reference this content in logging errors, events, and/or diagnostic messages. Ownership is passed on success, the user calls VXIStringDestroy( ) to free this when no longer required.
Value	Value name to cross-reference this content in logging errors, events, and/or diagnostic messages. Ownership is passed on success, the user calls VXIStringDestroy( ) to free this when no longer required.
stream	Handle for writing the content and closing the stream

SBlog Interface

SBlog defines extensions to the VXIlog interface that support the tag mechanism and the listener registration.

Function	Definition
vxilog	Include the VXIlog interface structure.
RegisterErrorListener	The given listener is notified for all errors.
UnregisterErrorListener	Unsubscribes the given listener.
RegisterDiagnosticListener	The given listener is notified for all diagnostic messages.
UnregisterDiagnosticListener	Unsubscribes the given listener.
ControlDiagnosticTag	Turn the diagnostic tag on (true) or off (false).
RegisterEventListener	The given listener is notified for all events.
RegisterEventListener	Unsubscribes the given listener.
RegisterContentListener	Subscribe the given listener for content write requests.
UnRegisterContentListener	Unsubscribes the given listener.

VXIlog

Include the VXIlog interface structure.

RegisterErrorListener

The given listener is notified for all errors (via VXIlog::Error or VError calls) as each calls is processed by SBlog. The call cannot be disabled.

```
VXIlogResult (*RegisterErrorListener)(
 struct SBlogInterface*    pThis,
 SBlogErrorListener*    alistener,
 void*    userdata
);
```

Parameter	Description
alistener	The subscribing Listener.
userdata	User data that is returned to the when notification occurs listener. Note: The same listener may be registered multiple times, as long as unique userdata is passed. In this case, the listener is called once for each unique userdata.

UnregisterErrorListener

Unsubscribes the given listener/userdata combination.

```
VXIlogResult (*UnregisterErrorListener)(
 struct SBlogInterface   * pThis,
 SBlogErrorListener*    alistener,
 void*    userdata
);
```

Parameter	Description
alistener	The subscribing Listener.
userdata	User data that was passed in during registration.

RegisterDiagnosticListener

The given listener is notified for all diagnostic messages (via VXIlog::Diagnostic or VDiagnostic calls) as each calls is processed by SBlog. ControlDiagnosticTag controls the set of Diagnostics that are returned to either true or false. By default all tags are assumed to be false when the listener is registered, and therefore off. A tag must be specifically turned to true (enabled) before any callbacks are generated. If a tag is enabled, all callbacks that registered on a given SBlog interface is invoked. Generally, only one callback is registered on each interface.

```
VXIlogResult (*RegisterDiagnosticListener)(
 struct SBlogInterface   * pThis,
 SBlogDiagnosticListener*    alistener,
 void*    userdata
);
```

Parameter	Description
alistener	The subscribing Listener.
userdata	User data that is returned to the when notification occurs listener. Note: The same listener may be registered multiple times, as long as unique userdata is passed. In this case, the listener is called once for each unique userdata.

UnregisterDiagnosticListener

Unsubscribes the given listener/userdata combination.

```
VXIlogResult (*UnregisterDiagnosticListener)(
 struct SBlogInterface    *pThis,
 SBlogDiagnosticListener*    alistener,
 void*    userdata
);
```

Parameter	Description
alistener	The subscribing Listener.
userdata	User data that was passed in during registration.

ControlDiagnosticTag

All diagnostic log tags are assumed to be off unless specifically enabled. This must be done by calling this function and setting the given tag to true.

```
VXIlogResult (*ControlDiagnosticTag)(
 struct SBlogInterface   * pThis,
 VXIunsigned    tagID,
 VXIbool    state
);
```

Parameter	Description
tagID	Identifier that classifies a group of logically associated diagnostic messages (usually from a single software module) that are desirable to enable or disable as a single unit. See the top of this file for tagID allocation rules.
state	Boolean flag to turn the tag on (true) or off (false).

RegisterEventListener

The given listener is notified for all events (via VXIlog::Event or VEvent calls) as each calls is processed by SBlog. The call cannot be disabled.

```
VXIlogResult (*RegisterEventListener)(
 struct SBlogInterface    *pThis,
 SBlogEventListener*    alistener,
 void*    userdata
);
```

Parameter	Description
alistener	The subscribing Listener.
userdata	User data that is returned to the when notification occurs listener. Note: The same listener may be registered multiple times, as long as unique userdata is passed. In this case, the listener is called once for each unique userdata.

UnregisterEventListener

Unsubscribes the given listener/userdata combination.

```
VXIlogResult (*UnregisterEventListener)(
 struct SBlogInterface*    pThis,
 SBlogEventListener*    alistener,
 void*    userdata
);
```

Parameter	Description
alistener	The subscribing Listener.
userdata	User data that was passed in during registration.

RegisterContentListener

The given listener is notified for all content write requests (via VXIlog::ContentOpen( ) calls) as each request is processed by SBlog. The call cannot be disabled.

```
VXIlogResult (*RegisterContentListener)(
 struct SBlogInterface   * pThis,
 SBlogContentListener*    alistener,
 void*    userdata
);
```

Parameter	Description
alistener	The subscribing Listener.
userdata	User data that is returned to the listener when notification occurs. Note: the same listener may be registered multiple times, as long as unique userdata is passed. In this case, the listener is called once for each unique userdata.

UnregisterContentListener

Unsubscribes the given listener/userdata combination.

```
VXIlogResult (*UnregisterContentListener)(
 struct SBlogInterface*    pThis,
 SBlogContentListener*    alistener,
 void*    userdata
);
```

Parameter	Description
alistener	The subscribing Listener.
userdata	User data that was passed in during registration.

SBlogListeners

These functions are used to register and unregister for log events. There are also convenience functions for enabling and disabling tags. Finally four logging listener functions are defined.

Function	Definition
SBlogListenerChannelData	Data that must be initialized then passed as the user data when registering and unregistering SBlogListeners.
SBlogListenerInit	Global platform initialization of SBlogListeners.
SBlogListenerShutDown	Global platform shutdown of SBlogListeners.
DiagnosticListener	SBlog listener for diagnostic logging.
ErrorListener	SBlog listener for error logging.
EventListener	SBlog listener for event logging.
ContentListener	SBlog listener for content logging.
ControlDiagnosticTag	Convenience function for enabling a diagnostic tag, identical to calling SBlogInterface::ControlDiagnosticTag.
RegisterErrorListener	Convenience functions for registering and unregistering an error listener, identical to calling SBlogInterface::RegisterErrorListener and SBlogInterface::UnregisterErrorListener except the user data must be a SBlogListener channel log data structure.
RegisterDiagnosticListener	Convenience functions for registering and unregistering a diagnostic listener, identical to calling SBlogInterface::RegisterDiagnosticListener and SBlogInterface::UnregisterDiagnosticListener except the user data must be a SBlogListener channel log data structure.
RegisterEventListener	Convenience functions for registering and unregistering an event listener, identical to calling SBlogInterface::RegisterEventListener and SBlogInterface::UnregisterEventListener except the user data must be a SBlogListener channel log data structure.
RegisterContentListener	Convenience functions for registering and unregistering a content listener, identical to calling SBlogInterface::RegisterContentListener and SBlogInterface::UnregisterContentListener except the user data must be a SBlogListener channel log data structure.

SBlogListenerChannelData

This is a structure, not a function. It gets passed to the Register/ Unregister[...]Listener( ) functions to indicate the OSR interface (SWIrec interface) and logical channel number for logging purposes.

```
SBlogListener (ChannelData)(
    channel,
    SWIrec
);
```

Parameter	Description
channel	Logical channel number for the resource.
SWIrec	SWIrec (OpenSpeech Recognizer) interface, if non-NULL events are logged to the SWIrec event log as well as to the SBlogListeners log file.

SBlogListenerInit

```
VXIlogResult SBlogListenerInit (
  const VXIchar* logFileName,
  VXIint32   maxLogSize,
  VXIbool   logToStdout,
  VXIbool   keepLogFileOpen
);
```

Parameter	Description
logFileName	Name of the file where diagnostic, error, and event information is written. Pass NULL to disable logging to a file.
maxLogSize	Maximum size of the log file, in bytes. When this size is exceeded, the log file is rotated, with the existing file moved to a backup name (.old) and a new log file started.
logToStdout	TRUE to log diagnostic and error messages to standard out, FALSE to disable. Event reports are never logged to standard out.
keepLogFileOpen	TRUE to keep the log file continuously open, FALSE to close it when not logging. Performance is much better if it is kept open, but doing so prevents system operators from manually rotating the log file when desired (simply moving it to a new name from the command line, with a new log file being automatically created).

SBlogListenerShutDown

A global platform shutdown of the SBlogListeners subsystem.

```
VXIlogResult SBlogListenerShutDown(void);
```

DiagnosticListener

This function gets registered via RegisterDiagnosticListener( ) to perform diagnostic logging to a file. This is only called by the SBlog subsystem to perform logging, never directly by users.

```
SBLOGLISTENERS_API void DiagnosticListener (
 SBlogInterface*    pThis,
 VXIunsigned   tagID,
 const VXIchar*   subtag,
 time_t   timestamp,
 VXIunsigned   timestampMsec,
 const VXIchar*   printmsg,
 void*   userdata
);
```

Parameter	Description
pThis	Pointer to the SBlogInterface that issued the callback.
tagID	Identifier that classifies a group of logically associated diagnostic messages (usually from a single software module) that are desirable to enable or disable as a single unit.
subtag	Arbitrary string that may be used to subdivide the diagnostic messages of that tagID, or provide additional standardized information such as the source file, function, or method. There are no rules for the content of this field.
timestamp	time_t the time of the log event.
timestampMsec	Milliseconds for the time of the log event.
printmsg	NULL terminated string to be printed.
userdata	User data that is delivered in the callback.

ErrorListener

This function gets registered via RegisterErrorListener( ) to perform error logging to a file. This is only called by the SBlog subsystem to perform logging, never directly by users.

```
SBLOGLISTENERS_API void ErrorListener (
 SBlogInterface*   pThis,
 const VXIchar*   moduleName,
 VXIunsigned   errorID,
 time_t   timestamp,
 VXIunsigned   timestampMsec,
 const VXIVector*   keys,
 const VXIVector*   values,
 void*   userdata
);
```

Parameter	Description
pThis	Pointer to the SBlogInterface that issued the callback.
moduleName	Name of the software module that is outputting the data.
errorID	Identifier that classifies a group of logically associated error messages (usually from a single software module) that are desirable to enable or disable as a single unit.
timestamp	time_t the time of the log event.
timestampMsec	Milliseconds for the time of the log event.
keys	Key name to cross-reference this content in logging errors, events, and/or diagnostic messages. Ownership is passed on success, the user calls VXIStringDestroy( ) to free this when no longer required.
values	Value name to cross-reference this content in logging errors, events, and/or diagnostic messages. Ownership is passed on success, the user calls VXIStringDestroy( ) to free this when no longer required.
userdata	User data that is delivered in the callback.

EventListener

This function gets registered via RegisterEventListener( ) to perform event logging to a file. This is only called by the SBlog subsystem to perform logging, never directly by users.

```
SBLOGLISTENERS_API void EventListener (
 SBlogInterface*   pThis,
 VXIunsigned   eventID,
 time_t   timestamp,
 VXIunsigned   timestampMsec,
 const VXIVector*   keys,
 const VXIVector*   values,
 void*   userdata
);
```

Parameter	Description
pThis	Pointer to the SBlogInterface that issued the callback.
eventID	Identifier that classifies a group of logically associated event messages (usually from a single software module) that are desirable to enable or disable as a single unit.
timestamp	time_t the time of the log event.
timestampMsec	Milliseconds for the time of the log event.
keys	Key name to cross-reference this content in logging errors, events, and/or diagnostic messages. Ownership is passed on success, the user calls VXIStringDestroy( ) to free this when no longer required.
values	Value name to cross-reference this content in logging errors, events, and/or diagnostic messages. Ownership is passed on success, the user calls VXIStringDestroy( ) to free this when no longer required.
userdata	User data that is delivered in the callback.

ContentListener

This function gets registered via RegisterContentListener( ) to perform content logging to a file. This is only called by the SBlog subsystem to perform logging, never directly by users.

```
SBLOGLISTENERS_API VXIlogResult ContentListener (
   SBlogInterface*   pThis,
   const VXIchar*   moduleName,
   const VXIchar*   contentType,
   void*   userdata,
   VXIString**   logKey,
   VXIString**   logValue,
   SBlogStream**   stream
);
```

Parameter	Description
pThis	Pointer to the SBlogInterface that issued the callback.
moduleName	Name of the software module that is outputting the data.
contentType	
userdata	User data that is delivered in the callback.
logKey	
logValue	
stream	

ControlDiagnosticTag

```
VXIlogResult ControlDiagnosticTag (
 VXIlogInterface*   pThis,
 VXIunsigned   tagID,
 VXIbool   state
);
```

Parameter	Description
tagID	Identifier that classifies a group of logically associated diagnostic messages (usually from a single software module) that are desirable to enable or disable as a single unit. See the top of SBlog.h for tagID allocation rules.
state	Boolean flag to turn the tag on (TRUE) or off (FALSE).

RegisterErrorListener

```
VXIlogResult RegisterErrorListener (
 VXIlogInterface*    pThis,
 SBlogErrorListener*    alistener,
 SBlogListenerChannelData*    channelData
);
```

Parameter	Description
alistener	The subscribing Listener.
channelData	User data that is returned to the when notification occurs listener. Note: the same listener may be registered multiple times, as long as unique userdata is passed. In this case, the listener is called once for each unique userdata.

RegisterDiagnosticListener

```
VXIlogResult RegisterDiagnosticListener (
 VXIlogInterface*    pThis,
 SBlogDiagnosticListener*    alistener,
 SBlogListenerChannelData*    channelData
);
```

Parameter	Description
alistener	The subscribing Listener.
channelData	User data that is returned to the when notification occurs listener. Note: the same listener may be registered multiple times, as long as unique userdata is passed. In this case, the listener is called once for each unique userdata.

RegisterEventListener

```
VXIlogResult RegisterEventListener (
 VXIlogInterface*    pThis,
 SBlogEventListener*    alistener,
 SBlogListenerChannelData*    channelData
);
```

Parameter	Description
alistener	The subscribing Listener.
channelData	User data that is returned to the when notification occurs listener. Note: the same listener may be registered multiple times, as long as unique userdata is passed. In this case, the listener is called once for each unique userdata.

RegisterContentListener

```
VXIlogResult RegisterContentListener (
 VXIlogInterface*    pThis,
 SBlogContentListener*    alistener,
 SBlogListenerChannelData*    channelData
);
```

Parameter	Description
alistener	The subscribing Listener.
channelData	User data that is returned to the when notification occurs listener. Note: the same listener may be registered multiple times, as long as unique userdata is passed. In this case, the listener is called once for each unique userdata.

Object interface

The OpenSpeech Browser invokes VoiceXML object elements through the VXIobject interface. This interface provides methods for executing arbitrary objects, permitting integrator specific extensions to the VoiceXML language.

The OpenSpeech Browser provides a basic VXIobject implementation. This implementation looks up and executes objects that are implemented as C/C++ code within the implementation. For each execution, it provides access to nearly all of the VXI interfaces for the channel in order to permit objects that perform advanced functionality such as telephony, recognition, prompting, ECMAScript execution, and internet access.

This implementation currently implements two objects: one that returns the object arguments as the result, and another that provides extended diagnostic logging via the VXIlog interface. Additional objects are added by modifying the implementation's object lookup table. See src/client/api/SBobject.cpp in the distribution, which is built into SBclient.dll.

The object interfaces are:

- ❏ VXIobject — Object interface.
- ❏ SBobject — SBobject implementation of VXIobject.

VXIobject

Abstract interface for VoiceXML object functionality which allows integrators to define VoiceXML language extensions that can be executed by applications through the VoiceXML object element. These objects can provide almost any extended functionality that the integrators want.

There is one object interface per thread/line.

Function	Definition
OBJECT_CLASS_ID	Keys identifying properties in VXIMap for Execute( ) and Validate( ).
OBJECT_VALUE	Keys identifying properties in the VXIMap for an individual parameter (element) in cases where “valuetype” is not “data”.
VXIobjectInterface	VXIobject interface for executing VoiceXML objects.

VXIobjectInterface

Function	Definition
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
(*Execute)	Execute an object.
(*Validate)	Validate an object, performing validity checks without execution.

(*Execute)

```
(*Execute)(
   struct VXIobjectInterface*    pThis,
   const VXIMap*    properties,
   const VXIMap*    parameters,
   VXIValue**    result
);
```

Parameter	Description
pThis	
properties	Map containing properties and attributes for the <object> as specified above.
parameters	Map containing parameters for the <object> as specified by the VoiceXML tag. The keys of the map correspond to the parameter name ("name" attribute) while the value of each key corresponds to a VXIValue based type. For each parameter, the interpreter evaluates any ECMAScript expressions. Then if the "valuetype" attribute is set to "ref" the parameter value is packaged into a VXIMap with three properties: ❑ OBJECT_VALUE: actual parameter value ❑ OBJECT_VALUETYPE: "valuetype" attribute value ❑ OBJECT_TYPE: "type" attribute value Otherwise a primitive VXIValue based type is used to specify the value.
result	Return value for the <object> execution, this is allocated on success, and the caller is responsible for destroying the returned value by calling VXIValueDestroy( ). The object's field variable is set to this value.

(*Validate)

```
(*Validate)(
   struct VXIobjectInterface*    pThis,
   const VXIMap*    properties,
   const VXIMap*    parameters
);
```

Parameter	Description
pThis	
properties	Map containing properties and attributes for the <object> as specified in the VoiceXML specification except that “expr” and “cond” are always omitted (are handled by the interpreter).
parameters	Map containing parameters for the <object> as specified by the VoiceXML tag. The keys of the map correspond to the parameter name ("name" attribute) while the value of each key corresponds to a VXIValue based type. See Execute( ) above for details.

SBobject

Provides an implementation of the VXIobject abstract interface for VoiceXML object functionality that allows integrators to define VoiceXML language extensions that can be executed by applications through the VoiceXML object element. These objects can provide almost any extended functionality that the integrators want.

There is one object interface per thread/line.

Function	Definition
SBobjectResources	Structure containing all the interfaces required for Objects.
SBobjectInit	Global platform initialization of SBobject.
SBobjectShutDown	Global platform shutdown of SBobject.
SBobjectCreateResource	Create a new object service handle.
SBobjectDestroyResource	Destroy the interface and free internal resources.

SBobjectResources

This structure must be allocated and all the pointers filled with created and initialized resources before creating the Object interface.

Function	Definition
log	Log interface
inet	Internet interface
jsi	ECMAScript interface
prompt	Prompt interface
rec	Recognizer interface
tel	Telephony interface

SBobjectInit

```
VXIobjResult SBobjectInit(
   VXIlogInterface*   log,
   VXIunsigned   diagLogBase
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.
diagLogBase	Base tag number for diagnostic logging purposes. All diagnostic tags for SBobject start at this ID and increase upwards.

SBobjectShutDown

```
VXIobjResult SBobjectShutDown(
   VXIlogInterface*   log
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.

SBobjectCreateResource

```
SBobjectCreateResource(
   SBobjectResources*    resources,
   VXIobjectInterface**    object
);
```

Parameter	Description
resources	A pointer to a structure containing all the interfaces that may be required by the object resource.
object	The returned <object> handler resource.

SBobjectDestroyResource

```
SBobjectDestroyResource(
   VXIobjectInterface** object
);
```

Parameter	Description
object	Destroy the interface and free internal resources. Once this is called, the resource interfaces passed to SBobjectCreateResource( ) may also be released.

Prompting interface

The OpenSpeech Browser plays prompts through the VXIprompt interface which provides methods for queueing and playing audio and TTS prompts. Prompt engine and platform specific properties can be passed for enhanced prompt control.

The OpenSpeech Browser implements the VXIprompt interface, using SpeechWorks Speechify for text-to-speech services. For fault tolerance, two or more Speechify servers may be configured, where a load sharing algorithm is used to distribute TTS requests across the Speechify server pool. Integrators typically do not need to modify or interact with this interface.

The OpenSpeech Browser VXIprompt implementation uses the VXIinet interface interface internally for fetching prompts.

The prompting interfaces are:

- VXIprompt — prompt interface
- SBprompt — SBprompt implementation of VXIprompt

VXIprompt

Abstract interface for Prompting functionality.

Function	Definition
VXIpromptResult	Result codes for interface methods.
VXIpromptInterface	Provides the interface to prompting.

VXIpromptResult

Function	Description
VXIprompt_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIprompt_RESULT_IO_ERROR	I/O error.
VXIprompt_RESULT_OUT_OF_MEMORY	Out of memory.
VXIprompt_RESULT_SYSTEM_ERROR	System error, out of service.
VXIprompt_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIprompt_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIprompt_RESULT_INVALID_PROP_ NAME	Property name is not valid.
VXIprompt_RESULT_INVALID_PROP_ VALUE	Property value is not valid.
VXIprompt_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIprompt_RESULT_SUCCESS	Success.
VXIprompt_RESULT_FAILURE	Normal failure, nothing logged.
VXIprompt_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIprompt_RESULT_FETCH_TIMEOUT	URL fetch time out.
VXIprompt_RESULT_FETCH_ERROR	URL fetch error.
VXIprompt_RESULT_BAD_SAYAS_CLASS	Bad sayas class.
VXIprompt_RESULT_TTS_ACCESS_ERROR	TTS access failure.
VXIprompt_RESULT_TTS_BAD_ DOCUMENT	TTS unsupported document type.
VXIprompt_RESULT_TTS_SYNTAX_ERROR	TTS document syntax error.
VXIprompt_RESULT_TTS_ERROR	TTS generic error.
VXIprompt_RESULT_RESOURCE_BUSY	Resource busy, such as TTS.
VXIprompt_RESULT_HW_BAD_TYPE	HW player unsupported MIME type.
VXIprompt_RESULT_HW_ERROR	Generic HW player error.
VXIprompt_RESULT_UNSUPPORTED	Operation is not supported.

VXIpromptInterface

Provides the interface to prompting.

Function	Description
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
BeginSession	Reset for a new session.
EndSession	Performs cleanup at the end of a call session.
Play	Start playing queued segments, non-blocking.
PlayFiller	Start the special play of a filler segment, non-blocking.
Prefetch	Prefetch a segment, non-blocking.
Queue	Queue a segment for playing, blocking.
Wait	Wait until all played segments finish playing, blocking.

BeginSession

```
VXIpromptResult (*BeginSession)(
 struct VXIpromptInterface    *pThis,
 VXIMap    *args
);
```

Parameter	Description
args	Implementation defined input and output arguments for the new session.

EndSession

```
VXIpromptResult (*EndSession)(
 struct VXIpromptInterface    *pThis,
 VXIMap                      *args
);
```

Parameter	Description
args	Implementation defined input and output arguments for ending the session.

PlayFiller

```
VXIpromptResult (*PlayFiller)(
 struct VXIpromptInterface    *pThis,
 const VXIchar    *type,
 const VXIchar    *src,
 const VXIchar    *text,
 const VXIMap             *properties,
 VXIlong    minPlayMsec
);
```

Parameter	Description
type	Type of segment, either a MIME content type, a sayas class name, or NULL to automatically detect a MIME content type (only valid when src is non-NULL).
src	URL or platform dependant path to the content, pass NULL when specifying in-memory text.
text	Text (possibly with markup) to play via TTS or sayas classes, pass NULL when src is non-NULL.
properties	Properties to control the fetch, queue, and play.
minPlayMsec	Minimum playback duration for the filler prompt once it starts playing, in milliseconds.

Prefetch

```
VXIpromptResult (*Prefetch)(
 struct VXIpromptInterface    *pThis,
 const VXIchar    *type,
 const VXIchar    *src,
 const VXIchar    *text,
 const VXIMap    *properties
);
```

Parameter	Description
type	Type of segment, either a MIME content type, a sayas class name, or NULL to automatically detect a MIME content type (only valid when src is non-NULL).
src	URL or platform dependant path to the content, pass NULL when specifying in-memory text.
text	Text (possibly with markup) to play via TTS or sayas classes, pass NULL when src is non-NULL.
properties	Properties to control the fetch, queue, and play.

Queue

```
VXIpromptResult (*Queue)(
 struct VXIpromptInterface    *pThis,
 const VXIchar    *type,
 const VXIchar    *src,
 const VXIchar    *text,
 const VXIMap    *properties
);
```

Parameter	Description
type	Type of segment, either a MIME content type, a sayas class name, or NULL to automatically detect a MIME content type (only valid when src is non-NULL).
src	URL or platform dependant path to the content, pass NULL when specifying in-memory text.
text	Text (possibly with markup) to play via TTS or sayas classes, pass NULL when src is non-NULL.
properties	Properties to control the fetch, queue, and play.

Wait

```
VXIpromptResult (*Wait)(
 struct VXIpromptInterface *pThis,
 VXIpromptResult           *playResult
);
```

Parameter	Description
playResult	Most severe error code resulting from a Play( ) operation since the last Wait( ) call, since Play( ) is asynchronous errors may have occurred after one or more calls to it have returned

SBprompt

SBprompt interface, an implementation of the VXIprompt abstract interface for Prompting functionality. The Prompt interface the handles prefetching, caching, and streaming audio as required to provide good response times and low CPU and network overhead.

Function	Definition
SBpromptInit	Global platform initialization of SBprompt.
SBrecpromptDown	Global platform shutdown of SBprompt.
SBrecpromptResource	Create a new prompt service handle.
SBrecpromptResource	Destroy the interface and free internal resources.

SBpromptInit

```
VXIpromptResult SBpromptInit (
 VXIlogInterface   *log,
 VXIunsigned   diagLogBase,
 const VXIchar   *recSrc,
 const VXIVector   *resources
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.
diagLogBase	Base tag number for diagnostic logging purposes. All diagnostic tags for SBprompt starts at this ID and increase upwards.
recSrc	Default recording source URI for concatenated prompt recording prompt type playback.
resources	Vector of prompting resources, may be NULL.

SBpromptShutDown

```
VXIpromptResult SBpromptShutDown (
 VXIlogInterface   *log
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.

SBpromptCreateResource

```
VXIpromptResult SBpromptCreateResource (
 VXIlogInterface    *log,
 VXIinetInterface    *inet,
 VXIcacheInterface    *cache,
 SBapcInterface    *aplay,
 VXIpromptInterface    **prompt
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging, must remain a valid pointer throughout the lifetime of the resource (until SBpromptDestroyResource( ) is called).
inet	VXI Internet interface used for URL fetches, must remain a valid pointer throughout the lifetime of the resource (until SBpromptDestroyResource( ) is called).
cache	VXI Cache interface used for write back caching, must remain a valid pointer throughout the lifetime of the resource (until SBpromptDestroyResource( ) is called).
aplay	SB Audio Player Control interface used for playing the audio, must remain a valid pointer throughout the lifetime of the resource (until SBpromptDestroyResource( ) is called).

SBpromptDestroyResource

Once this is called, the logging and Internet interfaces passed to SBpromptCreateResource( ) may also be released.

```
VXIpromptResult SBpromptDestroyResource (
 VXIpromptInterface    **prompt
);
```

Parameter	Description
prompt	Prompt interface.

Recognition interface

The OpenSpeech Browser does recognition through the VXIrec interface which provides non-blocking methods for loading grammars and blocking recognize and record methods. VXIrec allows passing recognition engine and platform specific properties for enhanced grammar management and recognition services.

The OpenSpeech Browser implements the VXIrec interface using the SpeechWorks OpenSpeech Recognizer. Integrators typically do not need to modify or interact with this interface.

The OpenSpeech Recognizer uses the VXIinet interface and VXIcache interfaces internally for fetching documents and caching compiled grammars. See the OpenSpeech Recognizer documentation for a complete description.

The recognition interfaces are:

- VXIrec — recognizer interface
- SBrec — SBrec implementation of VXIrec

VXIrec

Abstract interface for recognition functionality required by VoiceXML. Recognition is performed against VXIrecGrammars, abstract grammar types managed by the interface. The exact grammar formats handled are implementation dependant.

If an asynchronous problem/error occurs in the platform's recognizer, the VXI is notified through the VXIrecResult code for the current or next function call.

Function	Definition
VXIrec	Recognizer Interface.
VXIrecResult	Result codes for interface methods.

VXIrecResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Function	Definition
VXIrec_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIrec_RESULT_IO_ERROR	I/O error.
VXIrec_RESULT_OUT_OF_MEMORY	Out of memory.
VXIrec_RESULT_SYSTEM_ERROR	System error, out of service.
VXIrec_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIrec_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIrec_RESULT_INVALID_PROP_NAME	Property name is not valid.
VXIrec _RESULT_INVALID_PROP_VALUE	Property value is not valid.
VXIrec_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIrec_RESULT_SUCCESS	Success.
VXIrec_RESULT_FAILURE	Normal failure, nothing logged.
VXIrec_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIrec_RESULT_FETCH_TIMEOUT	URL fetch time out.
VXIrec_RESULT_FETCH_ERROR	URL fetch error.
VXIrec_RESULT_SYNTAX_ERROR	Grammar syntax error.
VXIrec_RESULT_HW_ERROR	Generic HW source error.
VXIrec_RESULT_UNSUPPORTED	Operation is not supported.

Keys identifying properties in VXIMap.

Properties	Definition
REC_BARGE_IN	Keys identifying properties in VXIMap.
VXIrecInputMode	Input modes as set in the REC_INPUT_MODE property defined above and as returned in VXIrecResult structures.
REC_MIME_GENERIC_DTMF	MIME content types for LoadGrammarURI and LoadGrammarString, the implementation usually supports additional MIME types.
VXIrecStatus	Status codes for recognition results.
REC_KEYS	Keys identifying properties in VXIMap used to return recognition result information for each n-best entry. See VXIrecRecognitionResult below for details.
VXIrecRecognitionResult	Recognition results structure as returned by Recognize.
VXIrecRecordResult	Record results structure as returned by Record.
VXIrecResult	Result codes for interface methods.
Recognition component interface	Provides the interface to the recognizer.

VXIrecInterface

Function	Definition
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
BeginSession	Reset for a new speaker.
EndSession	Performs cleanup at the end of a speaker’s session.
LoadGrammarURI	Load a grammar from a URI, typically non-blocking.
LoadGrammarString	Load an inline grammar, typically non-blocking.
ActivateGrammar	Activate a loaded VXIrecGrammar for subsequent recognition calls.
DeactivateGrammar	Deactivate a loaded VXIrecGrammar for subsequent recognition calls.
FreeGrammar	Free a loaded grammar.
Recognize	Recognize against the currently active grammars.
Record	Record an utterance.

BeginSession

```
VXIrecResult (*BeginSession)(
 struct VXIrecInterface    *pThis,
 VXIMap    *args
);
```

Parameter	Description
args	Implementation defined input and output arguments for the new session.

EndSession

```
VXIrecResult (*EndSession)(
 struct VXIrecInterface    *pThis,
 VXIMap    *args
);
```

Parameter	Description
args	Implementation defined input and output arguments for ending the session.

LoadGrammarURI

```
VXIrecResult (*LoadGrammarURI)(
 struct VXIrecInterface    *pThis,
 const VXIMap    *properties,
 const VXIchar          *type,
 const VXIchar          *uri,
 const VXIMap    *uriArgs,
 VXIrecGrammar    **gram
);
```

Parameter	Description
properties	Set of properties as defined above.
type	MIME type of the grammar, one of the REC_MIME_[...] defines above or an implementation defined grammar type.
uri	URI of the grammar definition.
uriArgs	Caching, timeout, base URI and other properties associated with URI retrieval.
gram	Handle to the new grammar.

LoadGrammarString

```
VXIrecResult (*LoadGrammarString)(
 struct VXIrecInterface    *pThis,
 const VXIMap    *properties,
 const VXIchar    *type,
 const VXIchar    *gramDef,
 VXIrecGrammar    **gram
);
```

Parameter	Description
properties	Set of properties as defined above.
type	MIME type of the grammar, one of the REC_MIME_ [...] defines above or an implementation defined grammar type.
gramDef	String containing the grammar definition.
gram	Handle to the new grammar.

ActivateGrammar

```
VXIrecResult (*ActivateGrammar)(
 struct VXIrecInterface    *pThis,
 const VXIMap    *properties,
 VXIrecGrammar    *gram
);
```

Parameter	Description
properties	Set of properties as defined above.
gram	Grammar to activate.

DeactivateGrammar

```
VXIrecResult (*DeactivateGrammar)(
 struct VXIrecInterface    *pThis,
 VXIrecGrammar    *gram
);
```

Parameter	Description
gram	Grammar to deactivate.

FreeGrammar

```
VXIrecResult (*FreeGrammar)(
 struct VXIrecInterface    *pThis,
 VXIrecGrammar    **gram
);
```

Parameter	Description
gram	Grammar to free, the pointer is set to NULL.

Recognize

```
VXIrecResult (*Recognize)(
 struct VXIrecInterface    *pThis,
 const VXIMap    *properties,
 VXIrecRecognitionResult    **recogResult
);
```

Parameter	Description
properties	Set of properties as defined above.
recogResult	Newly allocated result structure containing the result of the recognition. See the structure definition above.

Record

```
VXIrecResult (*Record)(
 struct VXIrecInterface    *pThis,
 const VXIMap     *properties,
 VXIrecRecordResult    **recordResult
);
```

Parameter	Description
properties	Set of properties as defined above.
recordResult	Newly allocated result structure containing the result of the record operation. See the structure definition above.

SBrec

SBrec interface, an implementation of the VXIrec abstract interface for recognition functionality.

Function	Definition
SBrecInit	Global platform initialization of SBrec.
SBrecShutDown	Global platform shutdown of Prompt.
SBrecCreateResource	Create a new prompt service handle.
SBrecDestroyResource	Destroy the interface and free internal resources.

SBrecInit

```
VXIrecResult SBrecInit(
 VXIlogInterface   *log,
 const VXIunsigned   diagLogBase
);
```

Parameter	Description
log	The logging interface used for the global pool.

SBrecShutDown

```
VXIrecResult SBrecShutDown(
 VXIlogInterface   *log
);
```

Parameter	Description
log	The logging interface used for the global pool.

SBrecCreateResource

```
VXIrecResult SBrecCreateResource(
 VXIlogInterface   *log,
 VXIrecInterface   **rec
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging, must remain a valid pointer throughout the lifetime of the resource (until SBrecDestroyResource( ) is called).
rec	VXIrecInterface pointer that is deallocated. It is set to NULL when deallocated.

SBrecDestroyResource

```
VXIrecResult SBrecDestroyResource(
 VXIrecInterface   **rec
);
```

Parameter	Description
rec	VXIrecInterface pointer that is deallocated. It is set to NULL when deallocated.

Telephony interface

The OpenSpeech Browser interacts with the telephony hardware for call control purposes through the VXItel interface. This interface provides methods for transferring and disconnecting calls. It does not provide methods for waiting for calls, as that is implementation dependant. VXItel allows passing telephony specific properties for enhanced telephony services.

The OpenSpeech Browser implements the VXItel interface using the VXIsessionControl interface, where most methods are simply passed through to a matching VXIsessionControl method. For example, the blind transfer method in the VXItel API is delegated down to the transfer method in VXIsessionControl. For this reason, integrators typically do not need to modify or interact with this interface.

The telephony interfaces are:

- VXItel — Telephony interface
- SBtel — SBtel implementation of VXItel

VXItel

VXItel provides the telephony functions for the VXI. The transfer type is split into the bridge and blind transfers. These platform functions are platform and location dependant.

Function	Definition
VXItelResult	Result codes for the telephony interface.
VXItelStatus	Telephony line status.
VXItelTransferStatus	Result codes for transfer requests.
VXItelInterface	Provides the interface to the telephony services.

VXItelResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Function	Definition
VXItel_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXItel_RESULT_DRIVER_ERROR	Low-level telephony library error.
VXItel_RESULT_IO_ERROR	I/O error.
VXItel_RESULT_OUT_OF_MEMORY	Out of memory.
VXItel_RESULT_SYSTEM_ERROR	System error, out of service.
VXItel_RESULT_PLATFORM_ERROR	Errors from platform services.
VXItel_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXItel_RESULT_INVALID_PROP_NAME	Property name is not valid.
VXItel_RESULT_INVALID_PROP_VALUE	Property value is not valid.
VXItel_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXItel_RESULT_SUCCESS	Success.
VXItel_RESULT_FAILURE	Normal failure, nothing logged.
VXItel_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXItel_RESULT_TIMEOUT	Operation is not supported.
VXItel_RESULT_UNSUPPORTED	Operation is not supported.

VXItelStatus

A line status of Active indicates that the line is currently occupied. It may be in a call or in a transfer.

Function	Definition
VXItel_STATUS_ACTIVE	In a call.
VXItel_STATUS_INACTIVE	Not in call.

VXItelTransferStatus

Result codes for transfer requests

Function	Definition
VXItel_TRANSFER_BUSY	The endpoint refused the call.
VXItel_TRANSFER_NOANSWER	There was no answer within the specified time.
VXItel_TRANSFER_NETWORK_BUSY	Some intermediate network refused the call.
VXItel_TRANSFER_NEAR_END_DISCONNECT	The call completed and was terminated by the caller.
VXItel_TRANSFER_FAR_END_DISCONNECT	The call completed and was terminated by the caller.
VXItel_TRANSFER_NETWORK_DISCONNECT	The call completed and was terminated by the network.
VXItel_TRANSFER_MAXTIME_DISCONNECT	The call duration exceeded the value of maxtime attribute and was terminated by the platform.
VXItel_TRANSFER_UNKNOWN	This value may be returned if the outcome of the transfer is unknown, for instance if the platform does not support reporting the outcome of blind transfer completion.

VXItelInterface

Provides the interface to the telephony services

Function	Definition
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
BeginSession	Reset for a new session.
EndSession	Performs cleanup at the end of a call session.
GetStatus	Queries the status of the line.
Disconnect	Immediately disconnects the caller on this line.
TransferBlind	Performs a blind transfer.
TransferBridge	Performs a bridge transfer.

BeginSession

This must be called for each new session, allowing for call specific handling to occur. For some implementations, this can be a no-op. For others runtime binding of resources or other call start specific handling may be done on this call.

```
VXItelResult (*BeginSession)(
 struct VXItelInterface   * pThis,
 VXIMap *   args
);
```

Parameter	Description
args	Implementation defined input and output arguments for the new session.

EndSession

This must be called at the termination of a call, allowing for call specific termination to occur. For some implementations, this can be a no-op. For others runtime resources may be released or other adaptation may be completed.

```
VXItelResult (*EndSession)(
 struct VXItelInterface   * pThis,
 VXIMap *   args
);
```

Parameter	Description
args	Implementation defined input and output arguments for ending the session.

GetStatus

Returns information about the line during an execution. Use to determine if the line is up or down.

```
VXItelResult (*GetStatus)(
 struct VXItelInterface   * pThis,
 VXItelStatus*    status
);
```

Parameter	Description
pThis	The line for which the status is to be queried.
status	A pointer to a pre-allocated holder for the status.

Disconnect

Disconnect the line. This sends the hardware into the out-of-service state where it no longer generates events.

```
VXItelResult (*Disconnect)(
 struct VXItelInterface   * pThis
);
```

Parameter	Description
pThis	The VXItel object for which the call is rejected.

TransferBlind

Perform a blind transfer into the network. The implementation is platform dependant and requires a good amount of configuration per installation location.

```
VXItelResult (*TransferBlind)(
 struct VXItelInterface   * pThis,
 const VXIMap*     properties,
 const VXIchar*     transferDestination,
     const VXIMap*     data,
 VXIMap **    resp
);
```

Parameter	Description
pThis	The line on which the transfer is to be performed.
properties	Termination character, length, timeouts...
transferDestination	Identifier of transfer location (e.g. a phone number to dial or a SIP URL).
data	The data to be sent to the transfer target.
resp	Key/value containing the result.

TransferBridge

Perform a bridge transfer into the network. The implementation is platform dependant and requires a good amount of configuration per installation location.

```
VXItelResult (*TransferBridge)(
 struct VXItelInterface*     pThis,
 const VXIMap*     properties,
 const VXIchar*      transferDestination,
 const VXIMap*     data,
 VXIMap**     resp
);
```

Parameter	Description
pThis	The line on which the transfer is to be performed.
properties	Termination character, length, timeouts...
transferDestination	Identifier of transfer location (e.g. a phone number to dial or a SIP URL).
data	The data to be sent to the transfer target.
resp	Key/value containing the result.

SBtel

SBtel provides an implementation of the VXItel abstract interface for call control functionality. All calls are blocking with the results returned by the implementation being used to determine the outcome of transfers. One VXItel interface should be constructed per line.

Function	Definition
SBtelInit	Initializes an SBtel implementation of the VXItel interface.
SBtelShutDown	Shutdown an SBtel implementation of the VXItel interface.
SBtelCreateResource	Creates an SBtel implementation of the VXItel interface.
SBtelDestroyResource	Destroys the specified SBtel implementation.

SBtelInit

```
VXItelResult SBtelInit (
 VXIlogInterface*   log,
 const VXIunsigned   diagLogBase
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.
diagLogBase	The logging base for the logging. All diagnostics are based on this offset.

SBtelShutDown

```
VXItelResult SBtelShutDown (
 VXIlogInterface  *log
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.

SBtelCreateResource

```
VXItelResult SBtelCreateResource(
 VXIscInterface  * sessionControl,
 VXIlogInterface*   log,
 VXItelInterface  ** tel
);
```

Parameter	Description
sessionControl	The session control object that is actually interacting with the hardware.
log	Logging interface used for error/diagnostic logging, must remain a valid pointer throughout the lifetime of the resource (until SBpromptDestroyResource( ) is called).
tel	A pointer used to hold the newly created SBtel object.

SBtelDestroyResource

```
VXItelResult SBtelDestroyResource(
 VXItelInterface  ** tel
);
```

Parameter	Description
tel	A pointer to the SBtel object to be destroyed.Set to NULL on return.

Thread library

The OpenSpeech Browser defines a set of C functions for basic mutex, thread, and timer functionality to ensure portability across a wide variety of operating systems and integration models (multi-threaded, multi-process, etc.). These functions are used throughout all the OpenSpeech Browser PIK component implementations and the reference code.

The OpenSpeech Browser implements the VXItrd library using a highly efficient operating system dependant implementation that is oriented towards a multi-threaded integration model. For a multi-process integration model, this library could be replaced with an alternate implementation that does not use threads, although it is usually better to simply perform direct kernel level locking and IPC within the integration components that communicate across processes or across systems (doing so may include building proxies for other OpenSpeech Browser PIK components that merely do IPC to another process that then invokes the original, unmodified OpenSpeech Browser PIK provided component).

The thread library is:

- ❏ VXItrd — thread library

VXItrd

C function library for basic mutex, thread, and timer functionality to ensure portability across a wide variety of operating systems and integration models (multi-threaded, multi-process, etc.).

Function	Definition
VXItrdResult	Result codes for functions.
VXItrdMutexCreate	Create a mutex.
VXItrdMutexDestroy	Destroy a mutex.
VXItrdMutexLock	Lock a mutex.
VXItrdMutexUnlock	Unlock a Mutex.
VXItrdThreadCreate	Create a thread.
VXItrdThreadDestroyHandle	Destroy a thread handle.
VXItrdThreadExit	Terminate a thread (called by the thread to exit).
VXItrdThreadJoin	Wait for the termination of a specified thread.
VXItrdThreadGetIDFromHandle	Get the thread ID for the specified thread.
VXItrdThreadGetID	Create a timer.
VXItrdTimerDestroy	Destroy a timer.
VXItrdTimerSleep	Suspend the current thread for a time period using a timer.
VXItrdTimerWake	Wakes a thread that is sleeping on a timer.

VXItrdMutexCreate

```
VXItrdResult VXItrdMutexCreate(
   VXItrdMutex**  mutex
);
```

Parameter	Description
mutex	Handle to the created mutex.

VXItrdMutexCreate

```
VXItrdResult VXItrdMutexDestroy(
   VXItrdMutex**  mutex
);
```

Parameter	Description
mutex	Handle to the mutex to destroy.

VXItrdMutexLock

```
VXItrdResult VXItrdMutexLock(
   VXItrdMutex*  mutex
);
```

Parameter	Description
mutex	Handle to the mutex lock.

VXItrdMutexUnLock

```
VXItrdResult VXItrdMutexUnLock(
   VXItrdMutex*  mutex
);
```

Parameter	Description
mutex	Handle to the mutex unlock.

VXItrdThreadCreate

Thread values are not supported on some older operating systems (such as the IBM OS/2). Execution starts on the thread immediately. To pause an execution, use a mutex between the thread and the thread creator.

```
VXItrdResult VXItrdThreadCreate(
   VXItrdThread**    thread,
   VXItrdThreadStartFunc    startFunc,
   VXItrdThreadArg    arg
);
```

Parameter	Description
thread	Handle to the thread that is created.
startFunc	Function for the thread to start execution on.
arg	Argument to the thread function.

VXItrdThreadDestroyHandle

This does NOT stop or destroy the thread; it just releases the handle for accessing it. If this is not done, a memory leak occurs, so if the creator of the thread never needs to communicate with the thread again it should call this immediately after the create if the create was successful.

```
VXItrdResult VXItrdThreadDestroyHandle(
   VXItrdThread**   thread
);
```

Parameter	Description
thread	Handle to the thread to destroy.

VXItrdThreadExit

```
void VXItrdThreadExit(
   VXItrdThreadArg   status
);
```

Parameter	Description
status	Exit value of the thread.

VXItrdThreadJoin

```
VXItrdResult VXItrdThreadJoin(
   VXItrdThread*   thread,
   VXItrdThreadArg*   status,
   long   timeout
);
```

Parameter	Description
thread	Handle to the thread to wait for.
status	Set to the exit value of the thread's start routine.
timeout	Timeout, in milliseconds, for waiting for the thread to exit, pass -1 to wait forever.

VXItrdThreadGetIDFromHandle

```
VXIlong VXItrdThreadGetIDFromHandle(
   VXItrdThread*  thread
);
```

Parameter	Description
thread	Handle to the thread to get the ID for.

VXItrdThreadGetID

Returns a platform-dependant thread identifier.

```
VXIlong VXItrdThreadGetID(void)
```

VXItrdThreadYield

Tells the operating system that the current thread wants to yield the CPU to other threads, causing a pause with the next thread in the CPU wait list getting activated.

```
void VXItrdThreadYield(void)
```

VXItrdTimerCreate

```
VXItrdResult VXItrdTimerCreate(
   VXItrdTimer**  timer
);
```

Parameter	Description
timer	Handle to create timer.

VXItrdTimerDestroy

```
VXItrdResult VXItrdTimerDestroy(
   VXItrdTimer**  timer
);
```

Parameter	Description
timer	Handle to the timer to destroy.

VXItrdTimerSleep

Due to other activities of the machine, the delay may be greater then the configured duration.

```
VXItrdResult VXItrdTimerSleep(
   VXItrdTimer*  timer,
   VXIint  sleepMs,
   VXIbool*  interrupted
);
```

Parameter	Description
timer	Handle to the timer to use to execute the suspend.
sleepMs	Duration to sleep, in milliseconds.
interrupted	Pointer indicating whether or not the sleep was interrupted by VXItrdTimerWake, TRUE if interrupted, FALSE if not. Pass NULL if this information is not desired.

VXItrdTimerWake

```
VXItrdResult VXItrdTimerWake(
   VXItrdTimer*  timer
);
```

Parameter	Description
timer	Handle to the timer to wake up.

Types and value library

The OpenSpeech Browser defines VXI prefixed primitive types to ensure portability across a wide variety of operating systems. These types are defined in VXItypes.h, and are used throughout all the VXI interfaces as well as within implementations of those interfaces.

In addition, the OpenSpeech Browser defines a higher-level data type library that provides more complex functionality such as vectors, maps, and reference counted BLOBs (binary large objects) along with a variant type that allows mixing those plus strings, integers, floating point numbers, and pointer values within those data structures. These are used throughout all the VXI interfaces to efficiently pass complex data such as VoiceXML property sets through those interfaces to permit value added VXI components without having to modify the VXI interfaces or the implementation of other components.

The OpenSpeech Browser implements the VXIvalue library using a highly efficient C++ implementation which is distributed both with the OpenSpeech Browser PIK as well as with the SpeechWorks OpenSpeech Recognizer. Integrators typically do not need to modify this library, but do need to understand and interact with this library as it is used throughout the OpenSpeech Browser interfaces and the reference code.

The types and value library is:

❑ VXItypes — VXI type definitions
❑ VXIvalue — Abstract VXI type library

VXItypes

VXI prefixed primitive types to ensure portability across a wide variety of operating systems. These types are used throughout all the VXI interfaces as well as within implementations of those interfaces.

Function	Definition
VXIptr	i386- *bindings.
VXI_CURRENT_VERSION	Current VXI interface version.
False	True and false for VXIbool values.

VXIvalue

Abstract run-time types for VXI interferences and optionally implementation. These types mirror ECMAScript types (and could be implemented as such). They could also be implemented using C++ and STL, or using C. Using these abstract types rather than directly using an externally defined class library increases portability, and allows the implementers of each interface to independently select external class libraries (or none at all) for their own implementation.

Each type is implemented as a handle that is obtained from a constructor and supports run-time typing. The owner (creator) of each type is responsible for freeing it by calling the appropriate destructor.

NOTE

When errors occur, constructors return a NULL object. Typically this is due to running out of memory, or a type mismatch for copy constructors.

The value types are as follows. Note that the naming convention is VXI followed by the type name starting with an uppercase letter, while the simple VXI types in VXIvalue.h use VXI followed by the type name starting with a lowercase letter:

Value types	Definition
VXIValue	Abstract base type for all the rest, can cast any type to this type and still determine the original type. Used to provide input/return values where the actual underlying type can vary.
VXIInteger	Container for a 32-bit integer (VXIint32).
VXIFloat	Container for a 32-bit float type (VXIflt32).
VXIString	Container for a string (VXIchar).
VXIPtr	Container for a untyped pointer (VXIptr).
VXIContent	Container for MIME content typed data (VXIptr).
VXIMap	Simple key/value container where the keys are of VXIchar type and the values are any of the abstract types defined here.
VXIVector	Simple indexed vector that supports appending elements at the end, and getting and setting elements by index. There is no support for removing elements or insertions.

Function	Definition
VXIvalueResult	Result codes for interface methods.
VXIValueGetType	Get the type of a Value.
VXIValueDestroy	Generic Value destructor.
VXIValueClone	Generic Value clone.
VXIIntegerCreate	Create an Integer from a 32 bit integer.
VXIIntegerDestroy	Integer destructor.
VXIIntegerValue	Get the value of an Integer.
VXIFloatCreate	Create a Float from a 32 bit floating point number.
VXIFloatDestroy	Float destructor.
VXIFloatValue	Get the value of a Float.
VXIPtrCreate	Create a Ptr from a C pointer.
VXIPtrDestroy	Ptr destructor.
VXIPtrValue	Get the value of a Ptr.
VXIContentCreate	Create a Content from MIME content typed data.
VXIContentDestroy	Content destructor.
VXIContentValue	Get the value of a Content.
VXIStringCreate	Create a String from a null-terminated character array.
VXIStringDestroy	String destructor.
VXIStringClone	String clone.
VXIStringSetValue	Set the value of a String from a null-terminated character array.
VXIStringValue	Get the value of a String.
VXIStringCStr	Get direct access to the NULL-terminated character value.
VXIStringLength	Get the number of characters in a String’s value.
VXIStringCompare	Compares two Strings.
VXIStringCompareC	Compares a String to a NULL-terminated character array.
VXIMapCreate	Create an empty Map.
VXIMapDestroy	Map destructor.
VXIMapClone	Map clone.
VXIMapSetProperty	Set a named property on an Map.
VXIMapGetProperty	Get a named property from an Map.
VXIMapDeleteProperty	Delete a named property from an Map.
VXIMapNumProperties	Return the number of properties for an Map.
VXIMapGetFirstProperty	Get the first property of an Map and an iterator.
VXIMapGetNextProperty	Get the next property of an Map based on an iterator.
VXIMapIteratorDestroy	Destroy an iterator.
VXIVectorCreate	Create an empty Vector.

Function	Definition
VXIVectorDestroy	Vector destructor.
VXIVectorClone	Vector clone.
VXIVectorAddElement	Adds an element to the end of the Vector.
VXIVectorSetElement	Set an indexed vector element.
VXIVectorGetElement	Get an indexed vector element.
VXIVectorLength	Return number of elements in a Vector.
VXIMapHolder	C++ wrapper class that makes it easier to work with VXIMaps.

Sample code

The following sample code illustrates the use of several of the VXIvalue based types, showing how to create a VXIMap that has a VXIInteger, a VXIFloat, and a VXIString as properties within it. All other VXIvalue types follow the same general model.

```
int main (int argc, char **argv)
{
   const VXIValue   *val;
   VXIMap   *myMap;
```

/* Construct a VXIMap (key/value pair container) which has:

1. Key named “myint”, integer value 1
2. Key named “myfloat”, floating point value 1.234
3. Key named “mystring”, string value “hello world”

*/

/* First allocate it */

```
myMap = VXIMapCreate( );
if ( myMap == NULL ) {
    fprintf (stderr, "ERROR: Out of memory\n");
    return 1;
}
```

/* Next add the data members. For each, we create another VXIvalue type and insert it, ownership is passed to the VXIMap such that when we destroy the map the properties (key/value pairs) are automatically destroyed too. */

```
if ( VXIMapSetProperty (myMap, L"myint",
     (VXIValue *) VXIIntegerCreate (1)) !=
       VXIvalue_RESULT_SUCCESS ) {
    fprintf (stderr, "ERROR: Out of memory\n");
    VXIMapDestroy (&myMap);
    return 1;
}

if ( VXIMapSetProperty (myMap, L"myfloat",
     (VXIValue *) VXIFloatCreate (1.234F)) !=
       VXIvalue_RESULT_SUCCESS ) {
    fprintf (stderr, "ERROR: Out of memory\n");
    VXIMapDestroy (&myMap);
    return 1;
}
if ( VXIMapSetProperty (myMap, L"mystring",
     (VXIValue *) VXIStringCreate (L"hello world")) !=
       VXIvalue_RESULT_SUCCESS ) {
    fprintf (stderr, "ERROR: Out of memory\n");
    VXIMapDestroy (&myMap);
    return 1;
}

/* Now get and display the contents of the map */
  val = VXIMapGetProperty (myMap, L"myint");
if (( val ) && ( VXIValueGetType (val) == VALUE_INTEGER ))
    printf ("myint = %d\n", VXIIntegerValue ((const
    VXIInteger *) val));
else
    fprintf (stderr, "ERROR: myint retrieval failed\n");
val = VXIMapGetProperty (myMap, L"myfloat");
if (( val ) && ( VXIValueGetType (val) == VALUE_FLOAT ))
    printf ("myfloat = %f\n", VXIFloatValue ((const VXIFloat
    *) val));
else
    fprintf (stderr, "ERROR: myfloat retrieval failed\n");

val = VXIMapGetProperty (myMap, L"mystring");
if (( val ) && ( VXIValueGetType (val) == VALUE_STRING ))
    printf ("mystring = %S\n", VXIStringCStr ((const
    VXIString *) val));
else
    fprintf (stderr, "ERROR: mystring retrieval failed\n");

/* Now destroy the map, freeing all the memory from the various Create( ) calls above
*/
  VXIMapDestroy (&myMap);

  return 0;
}
```

VXIvalueResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues).

Function	Definition
VXIvalueResult	Result codes for interface methods.
VXIvalue_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIvalue_RESULT_IO_ERROR	I/O error.
VXIvalue_RESULT_OUT_OF_MEMORY	Out of memory.
VXIvalue_RESULT_SYSTEM_ERROR	System error, out of service.
VXIvalue_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIvalue_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIvalue_RESULT_INVALID_PROP_NAME	Property name is not valid.
VXIvalue_RESULT_INVALID_PROP_VALUE	Property value is not valid.
VXIvalue_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIvalue_RESULT_SUCCESS	Success.
VXIvalue_RESULT_FAILURE	Normal failure, nothing logged.
VXIvalue_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIvalue_RESULT_UNSUPPORTED	Operation is not supported.

VXIValueGetType

```
VXIvalueType VXIValueGetType(
   const VXIValue*   v
);
```

Parameter	Description
v	Value to check.

VXIValueDestroy

This automatically invokes the appropriate type specific destructor.

```
void VXIValueDestroy(
   VXIValue**  v
);
```

Parameter	Description
v	Value to destroy.

VXIValueClone

This automatically invokes the appropriate type specific clone operation.

```
VXIValue VXIValueClone(
   const VXIValue*   v
);
```

Parameter	Description
v	Value to clone.

VXIIntegerCreate

```
VXIInteger*  VXIIntegerCreate(
   VXIint32 n
);
```

Parameter	Description
n	32 bit integer value.

VXIIntegerDestroy

```
void VXIIntegerDestroy(
   VXIInteger**  i
);
```

Parameter	Description
i	Integer to destroy

VXIIntegerValue

```
VXIint32 VXIIntegerValue(
   const VXIInteger*   i
);
```

Parameter	Description
i	Integer to obtain the value from.

VXIFloatCreate

```
VXIFloat* VXIFloatCreate(
   VXIflt32  n
);
```

Parameter	Description
n	32 bit floating point value.

VXIFloatDestroy

```
void VXIFloatDestroy(
   VXIFloat**  f
);
```

Parameter	Description
f	Float to destroy.

VXIFloatValue

```
VXIflt32 VXIFloatValue(
   const VXIFloat*   f
);
```

Parameter	Description
f	Float to get the value from.

VXIPtrCreate

This only stores the pointer blindly, it does not perform a deep copy and the reference memory is not freed on destruction. Thus the user is responsible for ensuring the referenced memory location remains valid, and for freeing memory when appropriate on Ptr destruction.

```
VXIPtr* VXIPtrCreate(
   void* n
);
```

Parameter	Description
n	Pointer to memory.

VXIPtrDestroy

```
void VXIPtrDestroy(
   VXIPtr**  p
);
```

Parameter	Description
p	Ptr to destroy

VXIPtrValue

```
void VXIPtrValue(
   const VXIPtr*  p
);
```

Parameter	Description
p	Ptr to retrieve the pointer from.

VXIContentCreate

Thread-safe reference counting is used to allow sharing the data (typically large) across multiple clones while minimizing memory use. The passed Destroy( ) function is only called when the reference count drops to zero.

```
VXIContent* VXIContentCreate(
   const VXIchar*    contentType,
   VXIbyte*    content,
   VXIulong    contentSizeBytes,
   void  (*Destroy)(VXIbyte**   content, void* userData),
   void*     userData
);
```

Parameter	Description
contentType	MIME content type for the data.
content	Data to store, this pointer is copied (no deep copy of the data is done) so this pointer must remain valid until the Destroy( ) function is called.
contentSizeBytes	Size of the data, in bytes.
Destroy	Destructor called to release the data when no longer needed. Since this construction merely copies the pointer, this is mandatory.
userData	Optional user data pointer passed to destroy, typically used to hold a pointer to some larger data structure that contains the content so that larger data structure can be destroyed when the content is no longer required.

VXIContentDestroy

```
void VXIContentDestroy(
   VXIContent**  c
);
```

Parameter	Description
c	Content to destroy.

VXIContentValue

```
VXIContentValue(
   const VXIContent*   c,
   const VXIchar**   contentType,
   const VXIbyte**   content,
   VXIulong*   contentSizeBytes
);
```

Parameter	Description
c	Content to retrieve the data from.
contentType	Returns the MIME content type for the data.
content	Returns the pointer to the data.
contentSizeBytes	Returns the size of the data, in bytes.

VXIStringCreate

```
VXIString* VXIStringCreate(
   const VXIchar*   str
);
```

Parameter	Description
str	NULL-terminated character array.

VXIStringCreateN

```
VXIStringCreateN(
   const VXIchar*   str,
   VXIunsigned   len
);
```

Parameter	Description
str	Character array (null characters may be embedded in the array).
len	Number of characters which are copied.

VXIStringDestroy

```
void VXIStringDestroy(
   VXIString**  s
);
```

Parameter	Description
s	String to destroy.

VXIStringClone

Functionally redundant with VXIValueClone( ), but provided to reduce the need for C casts for this common operation.

```
VXIString* VXIStringClone(
   const VXIString*   s
);
```

Parameter	Description
s	String to clone.

VXIStringSetValue

This functionality allows defining interfaces where the caller passes in a VXIString from VXIStringCreate( ) (typically with an empty string as its value) with the interface changing that value to return a string as output. This avoids having to define interfaces where the client has to provide a fixed length buffer (and thus worry about "buffer too small" errors and complicated handling).

```
VXIValueResult VXIStringSetValue(
   VXIString*   s,
   const VXIchar*   str
);
```

Parameter	Description
s	String to change the value of.
str	NULL-terminated character array.

VXIStringValue

```
VXIchar* VXIStringValue(
   const VXIString*   s,
   VXIchar*   buf,
   VXIunsigned   len
);
```

Parameter	Description
s	String to success.
buf	Character buffer to copy the value into as a NULL-terminated character array. The buffer size must be at least VXIStringLength( ) + 1.
len	Size of the buffer, in characters.

VXIStringCStr

The returned buffer must never be modified, and is only provided for transient use (i.e. immediately logging it, comparing it, etc. rather than storing or returning the pointer for longer term access).

```
const VXIchar* VXIStringCStr(
   const VXIString*   s
);
```

Parameter	Description
s	String to retrieve the data from.

VXIStringLength

Add one byte for the NULL terminator when using this to determine the length of the array required for a VXIStringValue( ) call.

```
VXIchar* VXIStringLength(
   const VXIString*   s
);
```

Parameter	Description
s	String to access.

VXIStringCompare

```
VXIint VXIStringCompare(
   const VXIString*   s1,
   const VXIString*   s2
);
```

Parameter	Description
s1	First string to compare.
s2	Second string to compare.

VXIStringCompareC

```
VXIint VXIStringCompareC(
   const VXIString*   str,
   const VXIchar*   buf
);
```

Parameter	Description
str	String to compare.
buf	NULL-terminated character array to compare.

VXIMapDestroy

This recursively destroys all the values contained within the Map, including all the values of Maps and Vectors stored within this map. However, for Ptr values the user is responsible for freeing the held memory if appropriate.

```
void VXIMapDestroy(
   VXIMap**   m
);
```

Parameter	Description
m	Map to destroy.

VXIMapClone

Copies all values contained within the map, including all the values of Maps and Vectors stored within this map.

Functionally redundant with VXIValueClone( ), but provided to reduce the need for C casts for this common operation.

```
VXIMap* VXIMapClone(
   const VXIMap*  m
);
```

Parameter	Description
m	Map to clone.

VXIMapSetProperty

The value can be an Map so a tree can be constructed.

If the property already exists, the existing value is first destroyed using VXIValueDestroy( ) (thus recursively deleting held values within it if it is an Map or Vector), then does the set operation with the new value.

```
VXIvalueResult VXIMapSetProperty(
   VXIMap*   m,
   const VXIchar*   key,
   VXIValue*   val
);
```

Parameter	Description
m	Map to access.
key	NULL terminated property name.
val	Value to set the property to, ownership is passed to the Map (a simple pointer copy is done), so on success the user must not delete, modify, or otherwise use this. Also do not add a Map as a property of itself (directly or indirectly), otherwise infinite loops may occur on access or deletion.

VXIMapGetProperty

The property value is returned for read-only access and is invalidated if the Map is modified. The client must clone it if they wish to perform modifications or wish to retain the value even after modifying this Map.

```
const VXIValue* VXIMapGetProperty(
   const VXIMap*    m,
   const VXIchar*    key
);
```

Parameter	Description
m	Map to access.
key	NULL terminated property name.

VXIMapDeleteProperty

This does a VXIValueDestroy( ) on the value for the named property (thus recursively deleting held values within it if it is an Map or Vector). However, for Ptr properties the user is responsible for freeing the held memory if appropriate.

```
VXIvalueResult VXIMapDeleteProperty(
   VXIMap*    m,
   const VXIchar*    key
);
```

Parameter	Description
m	Map to access.
key	NULL terminated property name.

VXIMapNumProperties

This only returns the number of properties that are direct children of the Map, it does not include the number of properties held in Maps and Vectors stored within this map.

```
VXIunsigned VXIMapNumProperties(
   const VXIMap*  m
);
```

Parameter	Description
m	Map to access.

VXIMapGetFirstProperty

This is used to traverse all the properties within an map, there is no guarantee on what order the properties are returned. The iterator must be eventually freed with VXIMapIteratorDestroy( ), and is invalidated if the Map is modified in any way.

```
VXIMapGetFirstProperty(
   const VXIMap*   m,
   const VXIchar**   key,
   const VXIValue**   value
);
```

Parameter	Description
m	Map to access.
key	Set to point at the property name for read-only access (must not be modified).
value	Set to point at the property value for read-only access (must not be modified).

VXIMapGetNextProperty

This is used to traverse all the properties within an map, there is no guarantee on what order the properties are returned.

```
VXIvalueResult VXIMapGetNextProperty(
   VXIMapIterator*   it,
   const VXIchar**   key,
   const VXIValue**   value
);
```

Parameter	Description
it	Iterator used to access the map as obtained from VXIMapGetFirstProperty( ), this operation advances the iterator to the next property on success.
key	Set to point at the property name for read-only access (must not be modified, invalidated if the Map is modified).
value	Set to point at the property value for read-only access (must not be modified, invalidated if the Map is modified).

VXIMapIteratorDestroy

```
void VXIMapIteratorDestroy(
   VXIMapIterator**   it
);
```

Parameter	Description
it	Iterator to destroy as obtained from VXIMapGetFirstProperty( ).

VXIVectorDestroy

This recursively destroys all the values contained within the Vector, including all the values of Vectors stored within this vector. However, for Ptr values the user is responsible for freeing the held memory if appropriate.

```
void VXIVectorDestroy(
   VXIVector**  v
);
```

Parameter	Description
v	Vector to destroy.

VXIVectorClone

Recursively copies all values contained within the vector, including all the values of Vectors and Maps stored within this vector.

Functionally redundant with VXIValueClone( ), but provided to reduce the need for C casts for this common operation.

```
VXIVector* VXIVectorClone(
   const VXIVector*   v
);
```

Parameter	Description
v	Vector to clone.

VXIVectorAddElement

The value can be a Vector so frames can be implemented.

```
VXIvalueResult VXIVectorAddElement(
   VXIVector*  v,
   VXIValue*  val
);
```

Parameter	Description
v	Vector to access.
val	Value to append to the vector, ownership is passed to the Vector (a simple pointer copy is done), so on success the user must not delete, modify, or otherwise use this. Do not add a Vector as a element of itself (directly or indirectly), otherwise infinite loops may occur on access or deletion.

VXIVectorSetElement

Overwrites the specified element with the new value. The existing value is first destroyed using VXIValueDestroy( ) (thus recursively deleting held values within it if it is an Map or Vector), then does the set operation with the new value.

The value can be a Vector so frames can be implemented.

```
VXIvalueResult VXIVectorSetElement(
   VXIVector*  v,
   VXIunsigned  n,
   VXIValue*  val
);
```

Parameter	Description
v	Vector to access.
n	Element index to set, it is an error to pass an index that is greater then the number of values currently in the vector.
val	Value to set the element to, ownership is passed to the Vector (a simple pointer copy is done), so on success the user must not delete, modify, or otherwise use this. Also do not add a Vector as a element of itself (directly or indirectly), otherwise infinite loops may occur on access or deletion.

VXIVectorGetElement

The element value is returned for read-only access and is invalidated if the Vector is modified. The client must clone it if they want to perform modifications or want to retain the value even after modifying this Vector.

```
const VXIValue* VXIVectorGetElement(
   const VXIVector*   v,
   VXIunsigned   n
);
```

Parameter	Description
v	Vector to access.
n	Element index to set, it is an error to pass an index that is greater or equal to then the number of values currently in the vector (i.e. range is 0 to length-1).

VXIVectorLength

This computes only the length of the Vector, elements within Vectors and Maps within it are not counted.

```
VXIunsigned VXIVectorLength(
   const VXIVector*   v
);
```

Parameter	Description
v	Vector to access.

VXI

The OpenSpeech Browser VoiceXML engine is implemented as the VXIinterpreter interface. This interface provides methods for configuring the VoiceXML interpreter and then for executing a VoiceXML application (document) against a specified set of interfaces that represent the channel. This interface also provides the functions required to initialize and destroy the VoiceXML interpreter.

The OpenSpeech Browser implementation of this interface is the open source SpeechWorks OpenVXI, and forms the core of the OpenSpeech Browser PIK. Integrators may substitute their own modified version of the OpenVXI if they wish to extend the VoiceXML language, although integrators are highly encouraged to do so via the VoiceXML object element and the VXIobject interface whenever possible.

The OpenSpeech Browser uses most of the other VXI interfaces to service VoiceXML document execution. The only exceptions are the VXI hardware interfaces, where the browser does not directly interface with them, instead those interactions are transparently handled by the VXIprompt, VXIrec, and VXItel implementations.

The VXI interpreter is:

❏ VXI — VXI interpreter interface.

VXI

The OpenSpeech Browser core is the OpenVXI. The VXIinterpreter interface implements the VXI interface function to run the interface. In addition a set of process and thread initialization routines are provided to set-up and destroy the interpreter per thread.

The structure of the VXI library functions is as follows:

Function	Description
VXI_BEEP_AUDIO	Keys identifying properties for SetProperties.
VXIinterpreterResult	Result codes for interface methods.
VXIresources	Structure containing all the interfaces required by the VXI.
VXIinterpreterInterface	VXIinterpreter interface for VoiceXML execution.
VXIinterpreterInit	Per-process initialization for VXIinterpreter.
VXIinterpreterShutDown	Per-process de-initialization for VXIinterpreter.
VXIinterpreterCreateResource	Create an interface to the VoiceXML interpreter.
VXIinterpreterDestroyResource	Destroy and de-allocate a VXI interface.

VXIinterpreterResult

Result codes less then zero are severe errors (likely to be platform faults), those greater then zero are warnings (likely to be application issues), with Success always being zero.

- If the errors are severe, call Technical Support.
- If the errors are warning, check your code.

Function	Definition
VXIinterp_RESULT_FATAL_ERROR	Fatal error, terminate call.
VXIinterp_RESULT_IO_ERROR	I/O error.
VXIinterp_RESULT_OUT_OF_MEMORY	Out of memory.
VXIinterp_RESULT_SYSTEM_ERROR	System error, out of service.
VXIinterp_RESULT_PLATFORM_ERROR	Errors from platform services.
VXIinterp_RESULT_BUFFER_TOO_SMALL	Return buffer too small.
VXIinterp_RESULT_INVALID_PROP_NAME	Property name is not valid.
VXIinterp_RESULT_INVALID_PROP_VALUE	Property value is not valid.
VXIinterp_RESULT_INVALID_ARGUMENT	Invalid function argument.
VXIinterp_RESULT_SUCCESS	Success.
VXIinterp_RESULT_FAILURE	Normal failure, nothing logged.
VXIinterp_RESULT_NON_FATAL_ERROR	Non-fatal non-specific error.
VXIinterp_RESULT_NOT_FOUND	Document not found.
VXIinterp_RESULT_FETCH_TIMEOUT	Document fetch timeout.
VXIinterp_RESULT_FETCH_ERROR	Other document fetch error.
VXIinterp_RESULT_INVALID_DOCUMENT	Not a VoiceXML document.
VXIinterp_RESULT_SYNTAX_ERROR	Document has syntax errors
VXIinterp_RESULT_UNCAUGHT_FATAL_ EVENT	Uncaught fatal VoiceXML event.
VXIinterp_RESULT_SCRIPT_SYNTAX_ERROR	ECMAScript syntax error.
VXIinterp_RESULT_SCRIPT_EXCEPTION	ECMAScript exception throw.
VXIinterp_RESULT_UNSUPPORTED	Operation is not supported.

VXIresources

This structure must be allocated and all the pointers filled with created and initialized resources before creating the VXI interface.

Function	Definition
log	Log interface.
inet	Inet interface.
jsi	ECMAScript interface.
rec	Recognizer interface.
prompt	Prompt interface.
tel	Telephony interface.
object	Object interface.
cache	Cache interface.

VXIinterpreterInterface

Abstract interface for the VoiceXML interpreter, simply provides a single method for running the interpreter on a document and getting the document result.

There is one interpreter interface per thread/line.

Function	Definition
GetVersion	Get the VXI interface version implemented.
GetImplementationName	Get the name of the implementation.
(*Run)	Run a VoiceXML document and optionally return the result.
(*SetProperties)	Specify runtime properties for the VoiceXML interpreter.
(*Validate)	Load and parse an VXML document.

Run

```
VXIinterpreterResult (*Run)(
   struct VXIinterpreterInterface*     pThis,
   const VXIchar*     name,
   const VXIMap*     sessionArgs,
   VXIValue**     result
);
```

Parameter	Description
name	Name of the VoiceXML document to fetch and execute, may be a URL or a platform dependant path. See the Open( ) method in VXIinet.h for details about supported names. However for URLs this must always be an absolute URL and any query arguments must be embedded.
sessionArgs	Any arguments to be passed to the VXI. Some of these, such as ANI, DNIS, etc. as required by VXML, but anything may be passed in. These values are available through the session variable in ECMA script.
result	(Optional, pass NULL if not desired.) Return value for the VoiceXML document (from), this is allocated on success and when there is an exit value (a NULL pointer is returned otherwise), the caller is responsible for destroying the returned value by calling VXIValueDestroy( ). If VXIinterp_RESULT_UNCAUGHT_ FATAL_EVENT is returned, this is a VXIString that provides the name of the VoiceXML event that caused the interpreter to exit.

SetProperties

```
VXIinterpreterResult (*SetProperties)(
   struct VXIinterpreterInterface*     pThis,
   const VXIMap*     props
);
```

Parameter	Description
props	Map containing a list of properties. Currently there are two of interest: ❏ VXI_BEEP_AUDIO URI for the beep audio and ❏ VXI_PLATFORM_DEFAULTS URI for the platform defaults

Validate

```
VXIinterpreterResult  (*Validate)(
   struct VXIinterpreterInterface*     pThis,
   const VXIchar*     name
);
```

Parameter	Description
name	Name of the VoiceXML document to fetch and execute, may be a URL or a platform dependant path. See the Open( ) method in VXIinet.h for details about supported names. However for URLs this must always be an absolute URL and any query arguments must be embedded.

VXIinterpreterInit

Per-process initialization for VXIinterpreter.

This function should be called once at process startup.

```
VXIinterpreterResult VXIinterpreterInit(
   VXIlogInterface*    log,
   VXIunsigned    diagLogBase
);
```

Parameter	Description
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.
diagLogBase	Base tag number for diagnostic logging purposes. All diagnostic tags for the VXI start at this ID and increase upwards.

VXIinterpreterShutDown

This function should be called once per process shutdown, after all the interfaces for the process are destroyed.

```
VXIinterpreterShutDown (
   VXIlogInterface*   log
);
```

Parameter	Definition
log	VXI Logging interface used for error/diagnostic logging. Only used for the duration of this function call.

VXIinterpreterCreateResource

Create a VXI interface given an interface structure that contains all the resources required for the VXI.

```
VXIinterpreterResult VXIinterpreterCreateResource(
   VXIresources*     resource,
   VXIinterpreterInterface**     pThis
);
```

Parameter	Definition
resource	A pointer to a structure containing all the interfaces requires by the VXI.
pThis	A pointer to the VXI interface that is to be allocated. The pointer is set if this call is successful.

VXIinterpreterDestroyResource

Destroy an interface returned from VXIinterpreterCreateResource. The pointer is set to NULL on success.

```
VXIinterpreterDestroyResource (
  VXIinterpreterInterface** pThis
);
```

Parameter	Definition
pThis	The pointer to the interface to be destroyed.

CHAPTER 5

Grammars

Inline grammars

The OpenSpeech Browser has primitive support for in-line Java( ) Speech API Grammar Format (JSGF) grammars in order to permit running most of the examples in the VoiceXML 2.0 specification. However, this capability is not a supported feature of the OpenSpeech Browser, so must not be used in production applications. (Note: Java™ is a trademark of Sun Microsystems Inc.). *[?Is this caveat still true?]*

Example ***[?Update this example for VXML 2.0]***

```
<?xml version="1.0" ?>
<vxml version="1.0">
<form id="form0">
  <field name="field0" >
<prompt>please vote for al gore, ralph nader or george
bush</prompt>
    <grammar > al_gore | ralph_nader | george_bush  </
grammar>
    <filled>
        <exit expr="field0" />
    </filled>
  </field>
</form>
</vxml>
```

You can inline W3C grammars using field and CDATA. For inline grammars, the type of grammar must be supplied in the voiceXML grammar type attribute. An example of an inline grammar is shown below.

Example

```
<?xml version="1.0" ?>
<vxml version="1.0" >
<form id="form0">
  <field name="field0">
    <grammar type="text/xml-grammar"> <CDATA>
    <grammar xml:lang="en-us" version="1.0" root="_boolean">

   <!--
    Copyright 2001 SpeechWorks International, Inc.
    All Rights Reserved.
    Boolean grammar
   -->

<meta name="swirec_max_speech_duration" content="7000" />
<meta name="incompletetimeout" content="500"/>
<meta name="swirec_compile_parser" content="1"/>

 <rule id="_boolean" scope="public">
    <one-of>
      <item tag="SWI_meaning=true; MEANING=SWI_
meaning;">yes</item>
      <item tag="SWI_meaning=true; MEANING=SWI_
meaning">correct</item>
      <item tag="SWI_meaning=false; MEANING=SWI_
meaning">no</item>
    </one-of>
 </rule>
</grammar>

    <CDATA/> <grammar/>
  </field>
</form>
```

URL grammars

The OSB fully supports the W3C grammar format through URLs. The URLs are loaded through the VXIinet interface by OSR and the support and information on how to write grammars for OSR is fully documented in the *OpenSpeech Recognizer User's* guide. *[?OSR-specific info]*

Applying grammar returns to field variables

W3C grammars return one or more semantic returns. *[?Update this link]*

See http://www.w3.org/TR/2000/NOTE-voicexml-20000505/#s10

Index