Mobile Conversay Software Development Kit

Getting Started

Conversational Computing Corporation
15375 NE 90th Street
Redmond, WA 98052

Conversay is interested in feedback about the quality and effectiveness of this document. Understanding your experiences and preferences will help us provide you with the highest level of support. We encourage you to share your valuable feedback with us via e-mail at Documentation_Feedback@conversay.com.

Lead Writer: Joe Perez

Contributors: Rob Barrows and Bruce Weber

Editor: Nelson Abbey

Documentation Manager: Sarah Mollet Baranowski

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 Introduction

Mobile Conversay™ Software Development Kit 1.02 is Conversay's voice platform for creating robust voice user interfaces on personal digital assistants, phones, and other mobile devices.

In This Section

- Features of Mobile Conversay SDK
- Overview
    - Documentation Conventions
    - Documentation Formats
- Developing in C
- Deploying on the Pocket PC Platform
- Deploying on the Embedded Linux Platform
- Other Resources

Features of Mobile Conversay SDK

Using Mobile Conversay SDK, a mobile application developer can create applications that perform the following tasks:

- Initialize and uninitialize multiple instances of speech recognition (SR) and text-to-speech (TTS) objects.
- Set callback events.
- Detect word, pause, speaking finished, and custom events.
- Recognize a specified grammar (U.S. English supported in this release).
- Use spelling-to-pronunciation (STP) rules to enhance pronunciation and recognition accuracy.
- Generate custom grammars, including multiple topics and contexts.
- Play a specified text string.
- Stop, fast forward, and rewind synthesized TTS output.
- Play .wav files on the audio output channel.
- Input .wav files for speech recognition.
- Detect the state of device microphone and speakers.
- Retrieve instances of user barge-in (only on full-duplex devices).
- Retrieve detailed error and troubleshooting information.

Overview

"Introduction" provides an overview of the SDK's features and information about the documentation. You will also learn how to access the API using the C language and how to distribute your application on Pocket PC and embedded Linux devices.

"Getting Started" describes the system requirements and installation procedures for integrating the Mobile Conversay SDK into your development environment. Descriptions are provided for the API header files, data files, sample applications, and libraries. You will also learn how to program a simple application that listens for specific words and phrases and responds with text-to-speech synthesis. The application also shows you how to play .wav files to provide helpful user feedback and help.

"Designing a Voice User Interface" describes principles and methods of designing voice user interfaces (VUIs) for speech recognition and text-to-speech applications. You will learn about the elements of a well-designed VUI and the differences between implicit, explicit, tapering, and incremental prompts. Tips are provided to help you avoid common mistakes in VUI design.

"BNF Grammar" describes Backus-Naur Form, a standard notation convention used to describe speech recognition grammars. Even if you are already familiar with BNF notation, you can find useful information such as a syntax reference and samples of common notations.

Documentation Conventions

The font and style conventions used in this document make it easy to identify items such as code to type, procedures, and cross-references to related topics.

Bold indicates a user interface element.

`Monospace font` indicates code, file names, and directory paths.

Finding Procedures

Procedure headings begin with the word “To” so you can find them quickly. Here is an example of a procedure:

To Find a Procedure

- Look for a heading that begins with the word “To.” Procedures with more than one step are numbered. One-step procedures like this one are marked with a bullet.

Finding Tips and Notes

Tips and notes appear with special formatting:

TIP: Tips provide recommendations that can help you increase the effectiveness of your programming.

NOTE: Notes contain additional information about features and techniques that are not part of a procedure.

Finding Cross-References to Related Topics

Cross-references to related sections of the SDK appear in the format illustrated below:

See Also: “Documentation Formats,” page 11

Documentation Formats

This document is available in .pdf, .html, and .chm formats. The .html and .chm help systems also include the contents of *A Guide to the CASSI Services API*, a document available as a separate .pdf file.

To View the .pdf Documentation

1. If you do not have Adobe Acrobat Reader software installed on your computer, download it from the Adobe Web site (www.adobe.com).
2. Open one or both of these files:
   a. `MobileConversaySDK_GettingStarted.pdf`

b. `MobileConversaySDK_API.pdf`

NOTE: Documentation is installed in the `docs` folder of the `Mobile Conversay SDK` folder. On Microsoft® Windows® systems, the default location is `C:\Program Files\Conversay\Mobile Conversay SDK\docs\`. On Linux systems, there is no default location; however, a typical location is `/usr/local/share/Conversay/MobileConversaySDK/docs/`.

To View the .html Documentation

- Open the file `MobileConversaySDK.html`.

NOTE: Documentation is installed in the `docs` folder of the `Mobile Conversay SDK` folder. On Windows systems, the default location is `C:\Program Files\Conversay\Mobile Conversay SDK\docs\`. On Linux systems, there is no default location; however, a typical location is `/usr/local/share/Conversay/MobileConversaySDK/docs/`.

To View the .chm Documentation

- Open the file `MobileConversaySDK.chm`.

NOTE: This help system is only available for Windows. Computers with operating systems older than Windows 2000 may require Microsoft Internet Explorer 4.0 or later to view the help file.

Developing in C

You can use the C or C++ programming languages to develop mobile applications with the Mobile Conversay SDK. This document provides syntax and examples in C++; however, the API supports C-style calls.

To program using C, it is necessary to use the `CVAOBJMACROS`. If `CVAOBJMACROS` is undefined or does not exist, then code written in C will not function. The `cvaservices.h` file, part of the standard API, contains the macros needed to translate C-style code into the corresponding C++ code.

An API call is presented below in three different ways. Then a full sample further illustrates the C-style calls.

C++ Code

```
ICVAObjectPtr->AddRef();
```

C Code Using C Style

```
ICVAObject_lpVtbl->AddRef(ICVAObjectPtr);
\* This call requires that *\
\* CVAOBJMACROS has been defined. *\
```

C Code Using C++ Style

```
ICVAObjectPtr->lpVtbl->AddRef(ICVAObjectPtr);
```

Full Sample

The sample below illustrates how an application can be designed using API calls in C. This sample shows conditional statements used to determine if `CVAOBJMACROS` has been defined. If the macro is defined, traditional C-style function calls are used; if the macro is not defined, API calls map a C++ style to a C structure.

```
#include <windows.h>

#define CVAOBJMACROS
#include "include\cvaservices.h"

int WINAPI WinMain( HINSTANCE hInstance,
     HINSTANCE hPrevInstance,
     LPTSTR    lpCmdLine,
     int       nCmdShow)
```

```
{
 ICVASRInstance *pSR;
 ICVASRContext *pContext;
 ICVATopic *pTopic;

// ***********************************************
// Creating instances of SR and context objects
// ***********************************************

 CVACreateInstance(CLSID_CVASRInstance, NULL, 0,
IID_ICVASRInstance, &pSR);
 CVACreateInstance(CLSID_CVAContext, NULL, 0,
IID_ICVAContext, &pContext);

// ***********************************************
// Adding a context to a SR object instance
// ***********************************************

#ifdef CVAOBJMACROS // If C macros defined...
 ICVASRInstance_Add(pSR, pContext);

#else // otherwise...
 pSR->lpVtbl->Add(pSR, pContext);
#endif

// Using class factory to create object instance
 CVACreateInstance(CLSID_CVATopic, NULL, 0,
IID_ICVATopic, &pTopic);

// ***********************************************
// Adding a topic to a context
// ***********************************************

#ifdef CVAOBJMACROS // If C macros defined...
 ICVAContext_Add(pContext, pTopic);

#else // otherwise...
 pContext->lpVtbl->Add(pContext, pTopic);
#endif

// ***********************************************
// Setting a grammar, compiling & activating topic
// ***********************************************

#ifdef CVAOBJMACROS // If C macros defined...
 ICVATopic_SetGrammar(pTopic, _T("<G> ::= ONE | TWO
| THREE | FOUR | FIVE | SIX | SEVEN | EIGHT | NINE
| ZERO | OH."), GRAM_FMT_BNF);
 ICVATopic_Compile(pTopic);
```

```
 ICVATopic_Activate(pTopic);

#else // otherwise...
 pTopic->lpVtbl->SetGrammar(pTopic, _T("<G> ::=
ONE | TWO | THREE | FOUR | FIVE | SIX | SEVEN |
EIGHT | NINE | ZERO | OH."), GRAM_FMT_BNF);
 pTopic->lpVtbl->Compile(pTopic);
 pTopic->lpVtbl->Activate(pTopic);
#endif
}
```

Deploying on the Pocket PC Platform

On the Pocket PC platform, you can deploy custom applications as well as the sample applications that are distributed with the SDK. CASSI™ Services is the API for accessing the core speech engine. The following files must be included in the system's `/windows/` directory for CASSI Services to function properly:

- `CVAServer.exe`
- `CVAAudio.dll`
- `CVAProxy.dll`
- `CVAPwrMgmt.exe`

CASSI, the core speech engine, performs speech recognition, synthesis, and text-to-speech. The following files must be included in the `/windows/` directory for CASSI to function properly:

- `C5cassi.dll`
- `c5ae08k.aqt`
- `c5ae08k.mod`
- `c5ae11k.spk`
- `c5ae.stp`
- `c5aemain.cdc`

Additionally, the following file is required for the Mobile Conversay environment:

- `SpPref.exe`

Deploying on the Embedded Linux Platform

On the embedded Linux® platform, you can deploy custom applications as well as the sample applications that are distributed with the SDK. To install applications on an embedded Linux device, CASSI dictionary and other files need to be transferred to the device. The following files must be included in the `/usr/local/share/cassi` directory or another directory specified in the `CASSI_HOME` environment variable:

- `c5ae08k.aqt`
- `c5ae08k.mod`
- `c5ae11k.spk`
- `c5ae.stp`
- `c5aemain.cdc`

Other Resources

Conversay strives to provide complete reference manuals and how-to guides, but occasional gaps occur. If you have trouble finding an answer in this documentation, then try these additional resources:

Sending Feedback

You can help to eliminate gaps in Conversay documentation in future releases. If you have suggestions or comments about this documentation, then we would like to hear from you. Please send your feedback to Documentation_Feedback@conversay.com.

Technical Support

For access to technical support, refer to your technical support contract with Conversay or a value-added reseller. If you do not have a technical support contract, call 1-888-487-4373 to purchase one.

The Conversay Web Site

Visit http://www.conversay.com for information about Conversay products, partners, and solutions.

The Conversay Developer Network

The Conversay Developer Network provides resources for developers who are creating and implementing Conversay-based speech technology solutions. To learn more about the CDN, visit http://cdn.conversay.com.

2 Getting Started

This section provides information on system requirements, installation, recognizing SDK files, and an introduction to programming well-behaved speech applications.

In This Section

- Supported Operating Systems
- Hardware Requirements
- Installing Mobile Conversay SDK on Windows
- Header Files
- Library Files
- Data Files
- Sample Applications
- Creating Well-Behaved Applications
    - Conversational Focus Management
    - Data Lifetime Control
    - Steps to Creating Well-Behaved Applications
    - Understanding the Basics: The Hello World Example

Supported Operating Systems

The SDK supports the following operating systems for developing your applications:

- Microsoft® Windows® 2000
- Linux® 2.4 kernel

The following embedded platforms and operating systems are supported for deploying applications:

- Pocket PC platform
- Embedded Linux 2.4 kernel

Hardware Requirements

The Mobile Conversay™ SDK supports most hardware configurations from all major Pocket PC devices equipped with microphones and speakers. The following chip sets are supported:

- StrongARM® microprocessor designed by ARM Ltd.
- SH3 (SuperH™ microprocessor) from Hitachi
- MIPS designed by MIPS Technologies
- 32-bit x86 platforms (Pocket PC emulator)

The following hardware configurations are supported on Linux platforms with microphones and speakers:

- StrongARM® microprocessor designed by ARM Ltd. (embedded systems)
- 32-bit x86 platforms (Linux desktop systems)

The following RAM is required to deploy the SDK:

- 16 MB RAM (32 MB recommended)

Installing Mobile Conversay SDK on Windows

To Install Mobile Conversay SDK on Windows 2000

1. Open Setup.exe. When you install from a CD-ROM, the program starts automatically as soon as you insert the disc into your CD-ROM player.
2. In the **Welcome** screen of the installation wizard, click **Install**.
3. Click **Next**.
4. Read the license agreement, select the **I Accept** option, and click **Next**.
5. Enter your user name and organization, select an access option, and click **Next**.
6. Select the **Complete** option, and click **Next**.
7. Click **Next**, click **Install**, and then click **Finish**.

NOTE: The default installation location is `C:\Program Files\Conversay\Mobile Conversay`.

To Build the Sample Applications on the Emulator:

- Use the Pocket PC emulator's Start Menu to launch the application.

NOTE: The Mobile Conversay SDK automatically installs all the needed files on the Pocket PC emulator.

To Install Sample Applications on Pocket PC Devices

- Copy the following files to the system's `/windows/` directory:

  `CVAServer.exe`

  `CVAAudio.dll`

  `CVAProxy.dll`

  `CVAPwrMgmt.exe`

  `C5cassi.dll`

  `c5ae08k.aqt`

  `c5ae08k.mod`

  `c5ae11k.spk`

  `c5ae.stp`

```
c5aemain.cdc
```

NOTE: Before building the samples, build the Portability library to ensure that the application functions on any supported operating system. To do this, open the `portability.vcp` file in the `samples/Portability` folder in Microsoft embedded Visual Studio, and then run Build All. In the sample's folder, open the .vcp file in Microsoft embedded Visual Studio and then run Build All.

Installing Mobile Conversay SDK on Linux

To Install Mobile Conversay SDK on Linux

1. Change directories to the directory where you wish to install the API (for example, `cd /usr/local/share`).
2. Verify that you have permission to write to the selected directory.
3. Copy the distributed .tar file to the selected directory (for example, `cp /tmp/CassiServicesLinuxBuild.tar.gz`).
4. Extract the file from the compressed .tar file (for example, `tar xzvf CassiServicesLinuxBuild25.tar.gz`).

   Two top level directories and sub-level directories are created. The top directories are `CASSIServices` and `CASSI`.
5. Before building anything, you must first set the environment variables. The file `CASSIServices/Examples/unix_bld_tools/CASSIServEnv.sh` contains examples of the variables that need to be set. Modify this file to point to the new location (for example, change `BASEDIR` to `BASEDIR=/usr/local/share`).
6. To place the environment variables into your current shell, run `. CASSIServices/Examples/bin/X86./CassiTest`.

NOTE: The `PATH` environment variable must contain the path to the `CASSIServices/bin/` and the `LD_LIBRARY_PATH` must contain the path to `CASSIServices/lib`/ for the platform you are running (for example, export `LD_LIBRARY_PATH=$LD_LIBRARY_PATH: /usr/local/Convseray/CASSIServices/X86`) and `PATH=$PATH:/usr/local/Convseray/CASSIServices/X86`).

To Build the Sample Applications on the Desktop:

1. `cd CASSIServices/Examples/Portability`
2. Type one of the following commands, depending on the chip set version you want to build:
   - `make X86` (X86 version).

- `make Linupy` (Linupy ARM).

NOTE: Only one platform can be built at a time and you must do a “make clean” between builds.

To Install Sample Applications on Embedded Linux Devices

1. Transfer the CASSI Dictionary and the other files in `CASSI/DataFiles` onto the device.

NOTE: By default, the `CASSIServices` library looks for the CASSI-related files in `/usr/local/share/cassi`. If you choose another location, you must set the `CASSIHOME` environment variable (e.g., export `CASSIHOME=/usr/local/Conversay/CASSI/Datafiles`).

2. Copy the files specific to the application onto the device’s `usr/local/bin` directory using `minicom` or some other means.

NOTE: To run the Hello World sample, it is necessary to copy the hellostart.wav and hellobye.wav files to the device’s `usr/local/bin` directory. These files are located in the `Examples/HelloWorld` directory.

Header Files

The following header files are needed to compile and link a speech application using the CASSI™ Services API.

Table 2-1 Header Files

File Name	Description
`CVAPtr.h`	Smart pointers for ICVA objects.
`CVAServices.h`	Interface declarations for all objects.
`CVATypes.h`	Public data type declarations for CASSI Services.
`STDTypes.h`	Additional type definitions.

Library Files

The following libraries must be included to create a speech application using the CASSI Services API. All of the .lib files have the same name (`CVAApi.lib`), but they are located in different folders.

Table 2-2 Library Folders

Folder	Platform
`ARMdbg`	Debug libraries for the StrongARM chip set.
`ARMrel`	Release libraries for the StrongARM chip set.
`SH3dbg`	Debug libraries for the SH3 chip set.
`SH3rel`	Release libraries for the SH3 chip set.
`X86EMdbg`	Debug libraries for the Pocket PC emulator.
`X86EMrel`	Release libraries for the Pocket PC emulator.

NOTE: Some of these file names are different for Linux systems.

When you create a speech application using the CASSI Services API, you must link to one of the library files provided in the `Lib` folder of the SDK. Each library file is created for a specific platform. The library file that you link to must correspond to the platform that you are creating the application on.

Data Files

Data files are provided in the `bin` directory and are used by CASSI at run time. The table below lists and describes the data files that are included with Mobile Conversay SDK:

Table 2-3 Data Folder

File	Description
`C5ae.stp`	*Spelling-to-pronunciation (STP) file.* Provides rules for postulating the pronunciation of any words that are not found in the dictionary or in auxiliary lexicons.
`C5ae08k.aqt`	*Acoustic question table file*. Required for compiling topics, speech recognition, and TTS.
`C5ae08k.mod`	*Acoustic model file.* Compares incoming speech to known acoustic segments. Required for speech recognition.
`C5ae11k.spk`	*Speaker file.* Gives acoustic values for text-to-speech output.
`C5aemain.cdc`	*Dictionary file.* Contains pronunciations for many common words. Required for compiling topics, speech recognition, and TTS.

NOTE: In the file names, “ae” denotes American English.

Sample Applications

The table below lists and describes the sample applications that are included with the SDK.

Table 2-4 Sample Applications

Name	Description
`FinancialApp`	A loan calculator program.
`HelloWorld`	A program that recognizes the spoken phrase “hello world” and responds with text-to-speech synthesis and audio output.

Creating Well-Behaved Applications

CASSI Services supports running multiple multi-threaded applications simultaneously on a mobile device. CASSI Services is designed to manage the system resources it uses and arbitrate demands for access to the audio system. However, application designers should still create well-behaved applications to avoid conflicts over audio resources and to avoid stretching other resources of the device beyond tolerable limits.

A well-behaved application manages conversational focus and exercises proper data lifetime control. In the topics that follow, you will learn about these techniques and the process used to make well-behaved applications. A complete application, “Hello World,” is given as an example to illustrate basic procedures and good techniques.

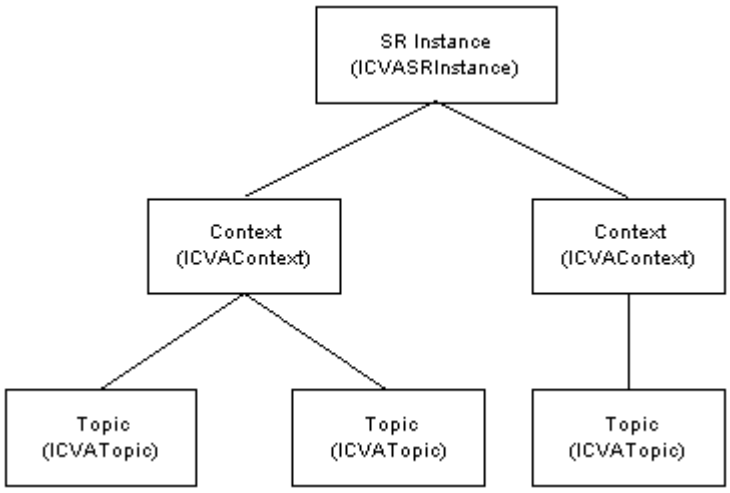
Conversational Focus Management

Conversational focus restricts speech recognition to designated elements within an application. While a speech application is running, conversational focus shifts depending on the tasks that are being performed. At any given time, some elements of the application will be “in conversational focus” and some will be “out of conversational focus.”

CASSI Services implements conversational focus management with the `ICVASRInstance`, `ICVAContext`, and `ICVATopic` interfaces (or simply “SR,” “context,” and “topic” objects). The SR object owns an object, or it can be the parent of one or more contexts. A context owns one or more topics. This hierarchical relationship is called the “focus hierarchy.” See Figure 2-1 on page 31.

An object is in focus if it is active and its parents are active. A topic is in focus if its SR is active, and its parent SR and context object are active. Likewise, a topic can be shifted “out of focus” by deactivating its context or SR instance or by deactivating its parent’s context or SR instance. Note that when a parent is deactivated and subsequently re-activated, its children are restored to their prior state, assuming that the children were not explicitly activated or deactivated in the interim.

Figure 2-1 Focus Hierarchy



Data Lifetime Control

A well-designed application optimizes performance and minimizes the amount of system resources that are used. One way to optimize performance is to eliminate unnecessary repetition of time-consuming functions like topic compilation. You should also avoid duplication of topics and grammars whenever possible. CASSI Services is designed to minimize resource usage by monitoring all of the API objects by a reference-count (or ref-count). When an object is no longer needed, its ref-count is decremented by *1*; when there are no longer any users of an object, its ref-count reaches *0* and it is automatically destroyed in order to free system resources.

To ensure that this mechanism functions properly, you must call the `Release` method on an object's interface pointer when you are done using it. Also, if you copy an interface pointer, it is essential that you call `AddRef` on the pointer prior to doing anything else with the original or the copy. For further information, see the resource management rules of the COM documentation.

When an application starts, it typically attempts to create some API objects. This attempt causes the CASSI Services server to load and start running if it is not running already. When no applications are using CASSI Services, the server is stopped and unloaded from memory. Each time an application creates an API object, CASSI Services checks available memory against the memory threshold and returns an error if there is not sufficient memory.

Steps to Creating Well-Behaved Applications

While every application is unique, certain steps are performed in a particular order in a typical well-behaved application:

Step 1. Creating a Speech Recognition Object

Step 2. Creating a Text-to-Speech Object

Step 3. Creating a Context Object

- To Create a Context Object
- To Add the Context to the SR Object

Step 4. Creating a Topic Object

- To Create a Topic
- To Add the Topic to the Context

Step 5. Setting and Compiling the Grammar

- To Set a Grammar for the Topic
- To Compile the Topic

Step 6. Setting the Callback Functions

Step 7. Controlling the Focus

- To Gain Control of Focus
- To Relinquish Control of Focus

Step 8. Processing Callback Events

Step 9. Releasing the Class Resources

- To Release The SR Object
- To Release The TTS Object

Step 1. Creating a Speech Recognition Object

A speech recognition (SR) object initiates the application to begin listening to the user's speech. The SR object is optional if your program does not require speech recognition.

To Instantiate a Speech Recognition Object

- Call `CVACreateInstance` with the interface identifier for the `ICVASRInstance` object.

Sample Code

```
CVACreateInstance(   // Instantiates an SR object
CLSID_CVASRInstance, // Class ID
NULL,                // NULL
0,                   // NULL
IID_ICVASRInstance,  // Interface ID
(void**)&m_pSR       // Pointer to object
);
```

Step 2. Creating a Text-to-Speech Object

Create a text-to-speech (TTS) object to enable the program to talk back to the user. The TTS object is optional if your program does not require text-to-speech synthesis.

To Instantiate a Text-to-Speech Object

- Call `CVACreateInstance` with the interface identifier for the `ICVATTSInstance` object.

Sample Code

```
CVACreateInstance(     // Instantiates TTS object
CLSID_CVATTSInstance,  // Class ID
NULL,                  // NULL
0,                     // NULL
IID_ICVATTSInstance,   // Interface ID
(void**)&m_pTTS        // Pointer to object
);
```

Step 3. Creating a Context Object

Create a context and add it to the SR object. At least one context is required, but you may create multiple contexts if necessary.

To Create a Context Object

- Call `CVACreateInstance` with the interface identifier for the `ICVAContext` object.

Sample Code

```
CVACreateInstance(
CLSID_CVAContext,     // Class ID
NULL,                 // NULL
0,                    // NULL
IID_ICVAContext,      // Interface ID
(void**)&m_pContext   // Pointer to object
);
```

To Add the Context to the SR Object

- Call the Add method of the `ICVASRInstance` object with the `ICVAContext` object.

Sample Code

```
m_pSR->Add(m_pContext // m_pSR is pointer to
);                    // the SR object
```

Step 4. Creating a Topic Object

Create a topic and add the topic to the context. At least one topic is required for each context. However, you may have multiple topics in each context.

To Create a Topic

- Call `CVACreateInstance` with the interface identifier for the `ICVATopic` object.

Sample Code

```
CVACreateInstance(
CLSID_CVATopic,     // Class identifier
NULL,               // NULL
0,                  // NULL
IID_ICVATopic,      // Interface identifier
(void**)&m_pTopic   // Pointer to object
);
```

To Add the Topic to the Context

- Call the Add method of the `ICVAContext` object with the `ICVATopic` object.

Sample Code

```
m_pContext->Add(m_pTopic // m_pContext is pointer
);                       // to the context object.
```

Step 5. Setting and Compiling the Grammar

Define the grammar and compile it to enable the topics to be recognized by the speech recognition engine. Use a text string in BNF format to supply the grammar.

To Set a Grammar for the Topic

- Call the `SetGrammar` method of the `ICVATopic` object with a text string.

Sample Code

```
m_pTopic->SetGrammar(
TEXT("<simplegrammar> ::= \
        HELLO_WORLD:HELLO | HOWDY_WORLD:HELLO | \
        HI_WORLD:HELLO | GOODBYE:BYE | \
        HELP:HELP."), GRAM_FMT_BNF);
```

To Compile the Topic

- Call the `Compile` method of the `ICVATopic` object.

Sample Code

```
m_pTopic->Compile(    // Compiles the grammar
);
```

Step 6. Setting the Callback Functions

Set a callback function that is fired when a speech recognition event occurs. If your application requires other events, such as knowing when an utterance is finished, you need to set callback functions for those events.

To Set a Callback for the Topic

- Call the `SetEventCallback` method of the `ICVATopic` object.

NOTE: Use the `SetEventCallback` method of the `ICVASRInstance` and `ICVAContext` objects to set callback functions for objects other than topic objects.

Sample Code

```
m_pTopic->SetEventCallback(
WORD_EVENT,    // Event type
this,          // Application data
Callback);     // Pointer to callback function
```

Step 7. Controlling the Focus

The next step is to establish the conversational focus to gain control of the device's audio system. Focus control is required when you have multiple contexts and topics. The contexts and topics must be activated and deactivated under various conditions to control what part of the program the user can interact with. Even in a very simple application with a single context and topic, the topic must still be activated and deactivated.

To Gain Control of Focus

- Call the `Activate` method of the `ICVATopic` object.

NOTE: The parent SR and context objects of the topic object must also be active for the topic to be put in focus.

Sample Code

```
m_pTopic->Activate( // Puts topic in focus
);
```

To Relinquish Control of Focus

- Call the `Deactivate` method of the `ICVATopic` object.

NOTE: A topic can also be put out of focus by deactivating its parent context or SR instance objects.

Sample Code

```
m_pTopic->Deactivate(  // Puts topic out of focus
);
```

Step 8. Processing Callback Events

The next step is to process your callback events. For example, you can synthesize a text-to-speech response or play a .wav file.

To Process Callback Events

1. Use conditional statements that allow you to detect each condition.
2. For each event you want to detect, call the `eventData` of the `EventMsg` structure.

Sample Code

```
if
(!_tcscmp(pEventMsg->eventData.RecoWord.pszTag,
TEXT("HELLO")))
```

```
{
   // Say an acknowledgement
   pThis->m_pTTS->Speak(
   TEXT("HI, HOW ARE YOU TODAY?"), false);
}
else
if
   (!_tcscmp(pEventMsg->eventData.RecoWord.pszTag,
   TEXT("BYE")))
{
   // Say an acknowledgement
   pThis->m_pTTS->Speak(
   TEXT("GOODBYE, HAVE A NICE DAY."), false);
}
break;
```

Step 9. Releasing the Class Resources

When your application is completed, release the class resources for any objects you have created.

To Release The SR Object

- Call the `Release` method of the `ICVASRInstance` object with the `ICVAContext` object.

Sample Code

```
m_pSR->Release(); // Releases the SR object
```

To Release The TTS Object

- Call the `Release` method of the `ICVASRInstance` object with the `ICVAContext` object.

Sample Code

```
m_pTTS->Release(); // Releases the TTS object
```

Understanding the Basics: The Hello World Example

The Hello World example is a complete application that demonstrates the CASSI Services speech recognition, text-to-speech, and audio streaming capabilities. The program plays a .wav file that prompts the user to say “hello world” or “goodbye.” The program listens for a response and offers a reply through text-to-speech syn-

thesis. The program terminates when the user says “goodbye.” You can find `helloworld.cpp` and the associated .wav files in the `samples` folder of the Mobile Conversay SDK.

Including the API Headers

This section of code includes the CASSI Services API files and a library of functions that are portable abstractions of operating system calls. These functions have names that begin with the word “Portable.” Code for these functions is in `portability.h` and `portability.cpp`.

Sample Code

```
#include "STDTypes.h"
#include "CVAServices.h"
#include "portability.h"
```

Defining Strings for OS Portability

This code defines strings that account for differences in the path-naming conventions on various platforms supported by the CASSI Services API.

Sample Code

```
#ifdef _WIN32
LPTSTR HelloStartFile =
TEXT("\\windows\\hellostart.wav");
LPTSTR HelloByeFile =
TEXT("\\windows\\hellobye.wav");
#endif
#ifdef unix
LPTSTR HelloStartFile =
TEXT("hellostart.wav");
LPTSTR HelloByeFile =
TEXT("hellobye.wav");
#endif
```

Defining the CHelloWorldApp Class

This code defines the class that implements simple speech recognition.

Sample Code

```
class CHelloWorldApp
{
     public:
     // The constructor
     CHelloWorldApp();
```

```
    // The destructor
    ~CHelloWorldApp();

    // Start the app, return when it exits
    CVAHRESULT Run();

    private:
    // Disable copy operator
    CHelloWorldApp(const
    CHelloWorldApp&);

    // Disable assign operator
    CHelloWorldApp& operator=(const
    CHelloWorldApp&);

    // Define the topic's callback
    static void Callback(const EventMsg*,
    void*, void*);

    // Release all the class resources
    CVAHRESULT ReleaseResources();

    // An event, when set, causes app to exit
    HANDLE m_exitEvent;

    // Interface to speech reco object
    ICVASRInstance* m_pSR;

    // Interface to speech synthesis object
    ICVATtsInstance* m_pTTS;

    // Interface to a context object
    ICVAContext* m_pContext;

    // Interface to a topic object
    ICVATopic* m_pTopic;

    // Interface to audio output stream object
    ICVAAudioOutStream* m_pOutStream;
};
```

Defining the Class Constructor

This code defines a method that is used to construct the `CHelloWorldApp` class.

Sample Code

```
CHelloWorldApp::CHelloWorldApp()
:m_exitEvent(0), m_pSR(0), m_pTTS(0),
```

```
m_pContext(0), m_pTopic(0), m_pOutStream(0)
{

}
```

Defining the Class Destructor

This code defines a method that is used to destroy the `CHelloWorldApp` class.

Sample Code

```
CHelloWorldApp::~CHelloWorldApp()
{

}
```

Defining the CHelloWorldApp:: Run Method

This code defines a method that is used to instantiate the speech recognition, text-to-speech, and audio objects. This method is also responsible for creating the context, topic, and grammar.

Sample Code

```
CVAHRESULT CHelloWorldApp::Run()
{
     CVAHRESULT res = CVAE_FAIL;
     // Create event to wait until time to exit
     m_exitEvent = PortableCreateEvent();
     if (!m_exitEvent)
{
     return CVAE_FAIL;
}
```

Creating the Speech Recognition Object

This code is part of the `CHelloWorldApp::Run` method. It instantiates the speech recognition object.

Sample Code

```
// Instantiate an SR object, leave it active
res = CVACreateInstance(CLSID_CVASRInstance,
NULL, 0, IID_ICVASRInstance, (void**)&m_pSR);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Creating the Text-to-Speech Object

This code is part of the `CHelloWorldApp::Run` method. It instantiates the text-to-speech object.

Sample Code

```
// Instantiate TTS object, leave it active
res = CVACreateInstance(CLSID_CVATTSInstance,
NULL, 0, IID_ICVATTSInstance, (void**)&m_pTTS);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Creating the Audio Output Object

This code is the part of the `CHelloWorldApp::Run` method. It instantiates the audio output stream object.

Sample Code

```
// Instantiate an audio output stream
// object, leave it active
res = CVACreateInstance(CLSID_CVAOutputStream,
NULL, 0, IID_ICVAOutputStream,
(void**)&m_pOutStream);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Creating a Context Object

This code is the part of the `CHelloWorldApp::Run` method. It instantiates the context object.

Sample Code

```
// Create a context, leave it active
res = CVACreateInstance(CLSID_CVAContext, NULL,
0, IID_ICVAContext, (void**)&m_pContext);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Adding the Context to the SR Object

This section of code is the part of the `CHelloWorldApp::Run` method. It adds the context object to the speech recognition object.

```
// Add the context to the SR object
res = m_pSR->Add(m_pContext);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Creating a Topic Object

This section of code is part of the `CHelloWorldApp::Run` method. It creates a topic object.

```
// Create a topic, leave inactive for now
res = CVACreateInstance(CLSID_CVATopic, NULL,
0, IID_ICVATopic, (void**)&m_pTopic);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Adding the Topic Object to the Context Object

This code is the part of the `CHelloWorldApp::Run` method. It adds the topic object to the context object.

```
// Add the topic to the context
res = m_pContext->Add(m_pTopic);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Setting a Grammar for the Topic

This section of code is part of the `CHelloWorldApp::Run` method. It sets the grammar for the topic object.

```
// Set a grammar for the topic using BNF
res = m_pTopic->SetGrammar(
TEXT("<simplegrammar> ::= \
        HELLO_WORLD:HELLO | HOWDY_WORLD:HELLO | \
        HI_WORLD:HELLO | GOODBYE:BYE | \
        HELP:HELP."),GRAM_FMT_BNF);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Compiling the Topic

This code is the part of the `CHelloWorldApp::Run` method. It compiles the grammar of the topic object.

```
// Compile the grammar
res = m_pTopic->Compile();
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Setting the Callback Function

This code sets the callback function.

Sample Code

```
// Set a callback for the topic
res = m_pTopic->SetEventCallback(WORD_EVENT,
this, Callback);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Activating the Topic

This code activates the topic.

Sample Code

```
// Activate the topic
res = m_pTopic->Activate();
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Playing the .wav Files

This code plays the `hellostart.wav` and `hellobye.wav` files.

Sample Code

```
// Play the "hellostart.wav" file
// This file contains the recording:
```

```
// "You can say hello world or you can say
// Goodbye."
res = m_pOutStream->SubmitFile(
HelloStartFile, false);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}

// Wait for the exit event
// When received,
// proceed with cleanup and return
PortableWaitOnEvent(m_exitEvent, INFINITE);

// Play the "hellobye.wav" file
// This file contains the recording:
// "Goodbye!"
res = m_pOutStream->SubmitFile(
HelloByeFile, false);
if (CVAFAILED(res))
{
     ReleaseResources();
     return res;
}
```

Calling the ReleaseResources() Function

This code calls the `ReleaseResources()` function.

Sample Code

```
// Release all resources used by the class
res = ReleaseResources();

return res;
}
```

Defining the CHelloWorldApp:: Callback Method

This code processes the callback function. It is called when a word or phrase in the topic's grammar is recognized. If the application recognizes the `HELLO`, `BYE`, or `HELP` grammar, it responds appropriately.

Sample Code

```
void CHelloWorldApp::Callback(const EventMsg*
pEventMsg, void* pvUserData, void* pv)
{
// Get the 'this' pointer (static member
// functions do not have direct access to it)
CHelloWorldApp* pThis =
```

```
(CHelloWorldApp*)pvUserData;

// Process according to the event type
switch(pEventMsg->eEventType)
{
     case WORD_EVENT:

     // Check the received word's tag
     // and respond accordingly
```

Recognizing the HELLO Grammar

This code is the case in the conditional statement that checks for the `HELLO` grammar and responds.

Sample Code

```
     if
     (!_tcscmp(pEventMsg-
>eventData.RecoWord.pszTag,
     TEXT("HELLO")))
{
     // Say an acknowledgement
     pThis->m_pTTS->Speak(
     TEXT("You said hello world."), false);
}
     else
```

Recognizing the BYE Grammar

This code is the case in the conditional statement that checks for the `BYE` grammar and returns an exit event.

Sample Code

```
     if
     (!_tcscmp(pEventMsg-
>eventData.RecoWord.pszTag,
     TEXT("BYE")))
{
     // Set the exit event
     // This will cause
     // run to unblock and return
     PortableSignalEvent(pThis->m_exitEvent);
}
     else
```

Recognizing the HELP Grammar

This code is the case in the conditional statement that checks for the `HELP` grammar and responds. Because this is a sample program, there is limited error handling. In an actual application, unexpected events would be handled also.

Sample Code

```
     if
     (!_tcscmp(pEventMsg-
>eventData.RecoWord.pszTag,
     TEXT("HELP")))
{
     // Play the "hellostart.wav" file
     // This file contains a recording:
     // "You can say hello world, or you can
     // say goodbye."
     pThis->m_pOutStream->SubmitFile(
     HelloStartFile, false);
}
     else
{
     // Unexpected phrase
}
break;
default:

// Unexpected event
break;
}
```

Defining CHelloWorldApp::ReleaseResources

This code releases the system resources used by the class.

Sample Code

```
CVAHRESULT CHelloWorldApp::ReleaseResources()
{
     CVAHRESULT res = CVAOK;
     CVAHRESULT returnVal = CVAOK;

// Remove the topic from the context
if (m_pTopic && m_pContext)
{
     res = m_pContext->Remove(m_pTopic);
     if (CVAFAILED(res))
{
returnVal = res;
}
}
```

```
// Remove the context from the SR object
if (m_pContext && m_pSR)
{
     res = m_pSR->Remove(m_pContext);
     if (CVAFAILED(res))
{
returnVal = res;
}
}

// Release the topic
if (m_pTopic)
{
     m_pTopic->Release();
     m_pTopic = 0;
}

// Release the context
if (m_pContext)
{
     m_pContext->Release();
     m_pContext = 0;
}

// Release the audio output stream
if (m_pOutStream)
{
     m_pOutStream->Release();
     m_pOutStream = 0;
}

// Release the SR object
if (m_pSR)
{
     m_pSR->Release();
     m_pSR = 0;
}

// Release the TTS object
if (m_pTTS)
{
     m_pTTS->Release();
     m_pTTS = 0;
}

// Delete the event object
if (m_exitEvent)
{
     PortableDeleteEvent(m_exitEvent);
```

```
     m_exitEvent = 0;
}

return returnVal;
}
```

Defining the Main Function

This code is the program's main function. It instantiates the `CHelloWorldApp` class and starts it running by calling `Run()`. When `Run()` returns, the program exits.

Sample Code

```
int __cdecl main(int  argc,  char  *argv[])
{
     CHelloWorldApp app;

     CVAHRESULT res = app.Run();
     // Won't return until commanded to do so
     if (CVAFAILED(res))
{
     return -1;
}

return 0;
}
```

3 Designing a Voice User Interface

This section provides an overview of the principles that you should consider when incorporating a voice user interface (VUI) into a graphical user interface (GUI).

In This Section

Designing For Speech

To be effective, a voice user interface (VUI) must provide a compelling benefit to your users. With that benefit in mind, design the application with speech in mind from the onset. Even if you are adding speech to an existing GUI, you should rethink the fundamental tasks from a conversational perspective.

Some tasks, easily represented in a GUI, may present challenges to represent in a VUI environment. For example, in a calendar application, finding exact dates is easy when the user can see a visual representation of the calendar and click the correct date. In the VUI environment, users think in relative terms. They may say things like “a week from yesterday” or “next Tuesday.” Understanding a user’s approach to a verbal task in relation to their approach to a visual task is an important principle in VUI design.

Prompts

Well-designed prompts are critical to the success of any VUI application. Prompts serve two purposes:

- As a cue when it is the user's turn to speak.
- As an indication of what may be spoken.

Because of this dual purpose, be careful to ensure that users can distinguish prompts from instructions or other non-interactive components. In addition, follow these tips as you design your prompts:

- Keep prompts short.
- Prompts should be preceded by instructions.
- Place important information immediately preceding the expected user response.

Explicit and Implicit Prompts

Prompts fall into two general categories: explicit and implicit. Explicit (also called directive) prompts indicate exactly what the user should say. Implicit prompts are open-ended; they do not list possible responses.

Example of an Explicit Prompt

System: Welcome to XYZ Brokerage. You can check an account balance, get a stock quote, or buy a stock. Say "check balance," "get quote," or "buy a stock."

Explicit prompts are useful in constraining user responses.

Example of an Implicit Prompt

System: Welcome to XYZ Brokerage. What would you like to do?

Implicit prompts are more conversational and can provide a natural interaction for the user. However, implicit prompts allow more room for user error.

Tapering Prompts

With tapering prompts you can make a repeated prompt shorter the second or third time that it is given. By removing unnecessary words or explicit directions, a more natural interchange is achieved.

Example of a Tapered Prompt

System: You have three new messages. The first is from Mark Adams. Say “read message,” “skip message,” or “delete message.”

You have two messages remaining. The next is from Mary Ruiz.

Incremental Prompts

Incremental prompts provide information to the user in small fragments. Each prompt is followed by a pause to allow for the user’s response. Incremental prompts can be a time saver for expert users. However, they can be problematic for novice users and can lead to collisions between the subsequent prompt and the user’s spoken response.

Example of an Incremental Prompt

System: Welcome to XYZ Brokerage. What would you like to do?

(Pause.)

You can check an account balance, get a stock quote, or buy a stock.

(Pause.)

Say “check balance,” “get quote,” or “buy a stock.”

See Also: “Latency,” page 59

Feedback

Feedback is a system output designed to inform users of the results of their actions. Feedback can be a visual cue, spoken in the form of text-to-speech (TTS), an auditory tone, or a combination of these elements. Feedback should provide users with the following information:

- Was their utterance heard?
- If heard, was the speech correctly interpreted?
- Is the system processing data or waiting for input?

Tips for Providing Feedback

Follow these tips when providing feedback to the user:

Implicitly Verify Commands That Present Data

Avoid literal feedback. Users get frustrated when a system constantly repeats, “You said...” and then re-states the exact response that it recognized. This is especially true if the system is wrong. Instead, implicitly verify commands by incorporating the verification into the next prompt. For example…

System: Which stock would you like to purchase?

User: XYZ Data.

System: How many shares of XYZ Data would you like to purchase?

Explicitly Verify Commands That Destroy Data or Are Irreversible

Be explicit when the cost of the action is high. For example…

System: Which contact would you like to delete?

User: Amy Wang.

System: To delete the contact for Amy Wang, say “yes” or “no.”

Provide a Transcription in Mixed Modal Systems

When combining VUI with GUI elements, provide a text transcription of what the user said. While literal feedback can be annoying to the user when provided via spoken system output, a text transcription can let the user know that the utterance was recognized.

Provide Other Visual Cues in Mixed Modal Systems

Visual cues, such as VU meters or icons, provide the user with feedback about the system. When working in a mixed modal system, it is best to provide visual cues about the system state whenever possible.

Dealing with Failures and Errors

CASSI™ recognizes a high percentage of user input, but occasional recognition failures and user errors do occur. It is up to the system designer to create an interface that accounts for recognition failures and minimizes the instances of user error.

Causes and Consequences of Failures and Errors

Common causes of failures and errors include the following:

- Hardware, such as microphone, turned off or not ready
- Background noise
- User spoke too soon
- Utterance not in grammar
- User paused too long during utterance
- Word is out of vocabulary

Failures and errors can have serious consequences in speech applications, breaking the user's perception of a human conversational model. While it is generally not a good idea to try to convince users that they are interacting with an almost-human machine, it is important to realize that users will frequently interact with the system as if it were human. When users speak to the system, they expect the system to respond.

Another consequence of a failure or error is that the user must initiate the correction. For example, if the user mispronounces a word and the system fails to recognize it, then the user must take action to correct the pronunciation.

Tips for Handling Failures and Errors

Follow these tips when handling system failures and user errors:

Do Not Use Repetitive Messages

The third or fourth time that an error prompt is repeated, it may lead to user frustration or anger, regardless of how politely the message is phrased.

Provide Progressive Assistance

Instead of just rephrasing the error prompt, incrementally provide more help to the user. For example…

Prompt 1: Say again.

Prompt 2: Sorry. Please rephrase.

Prompt 3: Still no luck. Please speak clearly, but do not overemphasize.

See Also: "Incremental Prompts," page 52

Provide a Model

One way of dealing with errors is to provide the user with a model. For example…

System: On what date would you like to make the new appointment?

User: A week from tomorrow.

System: I'm sorry. I don't understand. Say the date of the appointment. For example, say "April 23, 2001."

Provide Explicit Choices

Re-prompting with explicit choices can orient a user. For example…

System: Would you like to edit this contact?

User: I want the phone number.

System: I don't understand. Would you like to edit this contact? Say "yes" or "no."

Use Visual Cues if Available

Sometimes errors can be missed by the user. When designing for a mixed modal system, use visual cues to help the user recognize the occurrence of an error.

Mixing VUIs and GUIs

In general, the issues involved in designing prompts for mixed modal systems are the same as they are in VUI-only systems. Most importantly in mixed modal systems, you should keep in mind the ways that the user will access the interface.

For example, while some devices, such as cell phones, may be equipped with visual displays, you may be better off thinking in terms of voice only as you design such a VUI. When users interact with the voice interface of a cell phone, they may not be able to see the GUI. However, users of PDA devices may refer to the screen frequently.

If the screen is visible while the user speaks, then a GUI can be useful in providing feedback to the user. For example, a GUI can display a transcription of what the user has spoken, letting the user know if the system has recognized the command. GUIs can also provide a visual indication of the state of the system, such as whether or not the system is listening or processing an utterance.

Latency

In a VUI, pauses convey meaning to the user that you may not intend. System pauses that are inherent in the design of the system, often referred to as latency, can lead to misinterpretation by the user. For example, latency may be interpreted as a cue to speak. If your device is prone to system delays, then make sure to take these delays into account when designing your prompts.

4 BNF Grammar

This section of the documentation defines the grammar format, Backus-Naur Form (BNF), which is recognized by the Conversay™ speech engine.

In This Section

- About Backus-Naur Form
- Using BNF Grammars with CASSI
    - A Simple Grammar
    - Using Tag Mapping
    - Making Rules Optional
    - Using Recursive Grammars
    - Creating More Complex Grammars
    - Tips for Forming Grammars
- Example Grammars

About Backus-Naur Form

BNF was originally created to describe the syntax of the Algol 60 programming language. The convention was later modified and now describes the syntax of many languages. Easily learned and unambiguous, BNF defines a grammar with mathematical precision and is a broadly accepted standard. CASSI™, the speech engine of the Mobile Conversay API, also uses BNF.

Using BNF Grammars with CASSI

Grammars define the rules that a language must obey to be interpreted correctly. Speech recognition systems use grammars for an important reason: grammars allow the systems to achieve reasonable recognition accuracy and response time by constraining what the speech engine listens for. For example, a simple window control grammar may include phrases such as “open file,” “close window,” and “expand window.” The speech engine listens only for commands that make sense in the context of the window control grammar.

BNF uses a set of rules for terminal and non-terminal symbols. Terminal symbols, or characters, are words or phrases that the speech recognition system is listening for. Non-terminal symbols are bracketed by the < and > meta-symbols. Non-terminals are also called rules. Each rule in the grammar must have a corresponding terminal or another rule.

In a spreadsheet program, the grammar may include phrases like “new spreadsheet,” “open file,” “close file,” “save file,” “exit,” and “help.” In BNF format, this grammar is represented with this text string:

```
"<mainmenu> ::= NEW_SPREADSHEET | OPEN_FILE | \
CLOSE_FILE | SAVE_FILE | EXIT | HELP."
```

The definition follows the definition indicator (`::=`) and states that `<mainmenu>` is defined as “new spreadsheet,” “open file,” “close file,” “save file,” “exit,” and “help.” If `<mainmenu>` is the grammar listened for at the spreadsheet’s main menu interface, then the speech engine can listen for any of these phrases when the user visits the spreadsheet’s main menu. The definition indicator is a BNF meta-symbol that may be read as either “consists of” or “defined as.” The indicator always has a definition on its right-hand side.

In the example above, the `|` meta-symbol (read as “or”) separates different possible alternatives in a definition. The speech recognition engine will listen for any of the options separated by the `|` meta-symbol.

The `.` meta-symbol is used to indicate the end of a definition. You can include more than one definition in a text string by using the `.` meta-symbol to separate them.

The _ meta-symbol concatenates words in a phrase. For example, you may want the speech recognition engine to return “save file” rather than “save” and “file.”

NOTE: In C and C++ programs, BNF grammars are passed to the CASSI speech engine through text strings. Therefore, make sure to treat the BNF grammar as a text string by surrounding it with quotation marks (`""`) and concatenating more than one line with the `\` character.

Table 4-1 The Elements of BNF

Element	Appearance	Description
Terminal	`text`	Any character or character sequence that occurs in a text string and is a word or phrase that the speech recognition system is listening for.
Non-Terminal	`<rule>`	Any word in angle brackets. Must be defined somewhere within the string. In a speech grammar, the word in angle brackets is also called a rule.
Definition Indicator	`::=`	Used to define a terminal or non-terminal, which resides on the left side. The terminal or non-terminal on the right side is the definition.
Or Indicator	`|`	Separates alternatives in a definition.
Definition End Indicator	`.`	Indicates the end of a definition.

A Simple Grammar

Suppose we want to create a grammar that defines the syntax needed to look up the telephone area code of a city. The grammar could be part of a program that allows users to say an area code and get information about the place. To construct the BNF grammar, words and

phrases that users might say to find specific area codes are listed. For example, “area code,” “show me the area code,” and “where is the area code?”

To express a permissible sequence of words, the words are listed in the text string and separated by an underscore. The word pair `AREA_CODE` allows the phrase “area code” to be recognized, but not “code area.” These two words are terminals in BNF grammar, so they stand alone without further explanation. Terminals allow for any number of words in a sequence, such as `WHERE_IS_AREA_CODE`.

People typically use variations of phrases, depending on the individual person or context. To allow more than one word sequence, the grammar uses the or (`|`) indicator, like this:

```
"AREA_CODE | WHERE_IS_AREA_CODE | \
LOOK_UP_AREA_CODE."
```

The or indicator allows the program to recognize any of the three phrases. It is necessary in BNF notation to label such combinations of words and alternatives with a rule that describes how the user can use the phrases. This allows the program to refer to the alternatives many times in shorthand by a specified name. For instance, we could name the group of alternatives `start_phrase`:

```
"<start_phrase> ::= AREA_CODE | \
WHERE_IS_AREA_CODE | LOOK_UP_AREA_CODE."
```

NOTE: The definition indicator separates the defined word and is terminated with a `.` meta-symbol.

A grammar can consist of more than one rule. For example, the grammar that looks up the area code could continue like this:

```
"<area_code> ::= <digit> <digit> <digit>."
"<digit> ::= OH | ZERO | ONE | TWO | THREE | \
FOUR  | FIVE | SIX | SEVEN | EIGHT | NINE."
```

This simple grammar includes a rule, `<area_code>`, that is comprised of other rules. This rule structure ensures that only area codes comprised of three digits will be recognized.

Using Tag Mapping

Recall the previous example that showed phrases in the main menu of a spreadsheet program, such as “open file” and “exit.” While this gives users a way of performing tasks based on a certain phrase, you should design with the knowledge that users will use many different words or different phrases to perform a task. For example, a user might say “quit.” Since “quit” is not part of the grammar, the speech engine cannot recognize the word. Fortunately, this is an easy problem to avoid. It is possible to use a tag within the grammar to map more than one word to the same tag. The tag is placed to the right of the word and a (`:`) in the definition, like this:

```
"<mainmenu> ::= NEW_SPREADSHEET | OPEN_FILE |
CLOSE_FILE | SAVE_FILE | EXIT:EXIT | QUIT:EXIT |
HELP."
```

In the example above, the words “exit” and “quit” both have a tag of `EXIT`. It is possible to map any number of words or phrases to the same tag. Using multiple tag mapping relieves some of the burden of accuracy from the user. The grammar shown below offers more flexibility because of the multiple tag mapping:

```
"<hello_or_bye> ::= HELLO:HELLO | HOWDY:HELLO  |
HI:HELLO | GOODBYE:BYE | BYE:BYE."
```

The sample code includes a tag on each side of the `<hello_or_bye>` terminals. In this example, the tags allow the grammar to understand “hello,” “howdy,” and “hi,” as different ways of saying “hello.” Similarly, it recognizes “goodbye” and “bye,” as equivalent to “bye.”

Tags also allow the grammar to concatenate spoken digits into a single number, making the underlying processing more efficient. This is not a requirement because CASSI recognizes single digits by the words they are associated with, but may be useful in some applications. It may be desirable, for example, to make the following change to the grammar of the area code lookup program:

```
"<digit> ::= OH:0 | ZERO:0 | ONE:1 | TWO:2 |
THREE:3 | FOUR:4 | FIVE:5 | SIX:6 | SEVEN:7 |
EIGHT:8 | NINE:9."
```

With this grammar, if the user said “two oh one”, the tags attached to those digits could be used to concatenate the three numbers into the number *201*. If the user said “two zero one” or “two oh one”, the number *201* would be returned.

Making Rules Optional

The previous example showed that parts of a definition within a grammar can be made conditional by using tag mapping. Rules of a grammar can also be made optional. In BNF this is achieved by using the (|) indicator with the rule, similar to the way definitions are approached. Consider a grammar for a simple window control program without conditional rules:

```
"<command> ::= <action> <object>."
"<action> ::= OPEN | CLOSE | DELETE | MOVE."
"<object> ::= WINDOW | FILE | MENU."
```

This grammar allows the user to say commands such as “open window,” “close file,” or “move menu.” However, users may choose to say “open a file,” or “close the window.” Because “a” and “the” are not part of the grammar, these phrases are not recognized. It is possible to use tags to specify alternate phrases or even create many alternate definitions for these rules. However, these are not elegant or efficient solutions. Instead, the following grammar could be created:

```
"<command> ::= <action> <object> | <action>
<article> <object>."
"<action> ::= OPEN | CLOSE | DELETE | MOVE."
"<article> ::= THE | A."
"<object> ::= WINDOW | FILE | MENU."
```

In the example above, the words “the” and “a” define a rule called `<article>`. This rule is part of a definition for the `<command>` rule, but the definition for `<command>` also allows the user to speak without using “the” or “a.”

It’s a good idea to allow the user to perform speech commands using normal conversational phrases. This could include phrases such as “please.” Using optional rules allows users to be as polite as they want to be. Consider this grammar:

```
"<request> ::= <verb> <possession> <noun> |
<polite> <verb> <possession> <noun> | <verb>
<possession> <noun> <polite>."
"<verb> ::= GET | SEE | OPEN."
"<possession> ::= MY."
"<polite> ::= PLEASE."
"<noun> ::= CONTACTS | SCHEDULE | CALENDAR."
```

In this grammar, `<request>` can be recognized whether the user says “Get my schedule,” “Please get my schedule,” or “Get my schedule please.”

Using Recursive Grammars

Recursive grammars allow the user to say consecutive lists of words or items and have continuous recognition occur. For example, a user could say a number of indefinite length such as an address or a currency amount:

```
"<digits> ::= <digit> <digits> | <digit>."
"<digit> ::= ONE | TWO | THREE | FOUR | FIVE |
SIX | SEVEN | EIGHT | NINE | ZERO | OH:ZERO."
```

In addition, using recursive grammars, it is possible for a user to spell a word such as a stock symbol by speaking a word to represent each letter. For example, the user could say “kilo oscar” to indicate the symbol for the Coca-Cola Company, “KO.”

```
"<stock_symbol> ::= <letter><letters> | <letter>."
"<letters> ::= <letter><letters> | <letter>."
"<letter> ::= ALPHA:A |BRAVO:B | CHARLIE:C |
"DELTA:D | ECHO:E | FOXTROT:F | GOLF:G | HOTEL:H
| INDIA:I | JULIET:J | KILO:K | LIMA:L | MIKE:M
| NOVEMBER:N | OSCAR:O | PAPA:P | QUEBEC:Q |
ROMEO:R | SIERRA:S | TANGO:T | UNIFORM:U |
VICTOR:V | WHISKEY:W | X-RAY:X | YANKEE:Y |
ZULU:Z."
```

Creating More Complex Grammars

Using optional rules and recursion it is possible to create complex grammars. For example, consider a grammar for the Preamble to the Constitution of the United States of America. The grammar could be used to create a quiz program that prompts students to recite the entire 52-word phrase by memory, phrase by phrase, starting with the opening phrase. If desired, the quiz program could be designed to distinguish between a part of a phrase that was correct and a phrase that was almost correct, but not quite. It would also be possible for the program to offer suggestions for each phrase that the student missed. For example, if a student said “in order to form a perfect union,” the program could respond with “You are very close. Don’t give up now. Try again.” Or, the program could say “Try again, but

this time say ‘form a *more* perfect union’ instead of ‘form a perfect union.” The program could continue in this fashion until the student recited the entire Preamble.

The problem of composing a grammar that could be used for the quiz program can be tackled in three steps. The first task is to assign each part of the sentence a rule name; the name could be based on the phrase’s part of speech, such as “subject” or “verb.” For each sentence part, two rules are created: one with the label of `_exact` to indicate the exact word or phrase, and another with the label `_almost` for very close answers. A complete set of these rules would look like this:

```
"<subject_exact> ::= WE_THE_PEOPLE."
"<subject_almost> ::= WE | WE_PEOPLE."

"<subj_modifier_exact> ::=
OF_THE_UNITED_STATES_OF_AMERICA."
"<subj_modifier_almost> ::= OF_THE_U_S_A."

"<prep_phrase_exact> ::= IN_ORDER_TO."
"<prep_phrase_almost> ::= SO_THAT | SO_WE_CAN."

"<prep_phrase_1_exact> ::=
FORM_A_MORE_PERFECT_UNION."
"<prep_phrase_1_almost> ::= FORM_A_PERFECT_UNION."

"<prep_phrase_2> ::= ESTABLISH_JUSTICE."

"<prep_phrase_3_exact> ::=
INSURE_DOMESTIC_TRANQUILITY."
"<prep_phrase_3_almost> ::=
INSURE_DOMESTIC_HARMONY | INSURE_TRANQUILITY."

"<prep_phrase_4> ::=
PROVIDE_FOR_THE_COMMON_DEFENSE."

"<prep_phrase_5> ::= PROMOTE_THE_GENERAL_WELFARE."

"<prep_phrase_6a> ::=
SECURE_THE_BLESSINGS_OF_LIBERTY."

"<prep_phrase_6b_exact> ::=
TO_OURSELVES_AND_OUR_POSTERITY."
"<prep_phrase_6b_almost> ::=
TO_OUR_POSTERITY_AND_OURSELVES."

"<verb_phrase_exact> ::= DO_ORDAIN_AND_ESTABLISH."
```

```
"<verb_phrase_almost> ::= DO_ESTABLISH_AND_ORDAIN
| ORDAIN_AND_ESTABLISH."

"<pred_adj_exact> ::= THIS."
"<pred_adj_almost> ::= THE."

"<pred_obj> ::= CONSTITUTION."

"<pred_phrase_exact> ::=
FOR_THE_UNITED_STATES_OF_AMERICA."
"<pred_phrase_almost> ::= FOR_THE_U_S_A |
FOR_THE_UNITED_STATES."
```

The next step in defining the grammar would be to create additional rules that can be used to ascertain if the student has said a given phrase closely, but not exactly. One way to do this would be to use recursion to create additional rules such as these:

```
"<opening_phrase_exact> ::= <subject_exact>
<subj_modifier_exact>."
"<opening_phrase_almost> ::= <subject_almost>
<subj_modifier_almost>."
"<subject_almost> ::= <subject_exact> |
<subject_almost>."
"<subj_modifier_almost> ::= <subj_modifier_exact>
| <subj_modifier_almost>."
```

The above rules establish that several alternative phrases to “We the people of the United States of America” will be recognized as almost like the opening phrase of the Preamble. For example, “We the people of the USA,” “We people of the USA,” and “The people of the United States of America” are all defined as almost the opening phrase. However, since any correct grammar for the opening phrase consists in the subject followed by the subject modifier, nonsense phrases will not be mistaken for close expressions. For example, if a student said “Of the United States of America, we the people” this would not be recognized.

After rules such as these have been established for each phrase in the Preamble, the grammar could be completed by establishing rules for what defines a completely correct expression and what is considered a close approximation. For example, the rules that complete the grammar could look like these:

```
"<preamble_exact> ::= <subject_exact>
<subj_modifier_exact> <prep_phrase_exact>
<prep_phrase_1_exact> <prep_phrase_2>
<prep_phrase_3_exact> <prep_phrase_4>
```

```
<prep_phrase_5> <prep_phrase_6a>
<prep_phrase_6b_exact> <verb_phrase_exact>
<pred_adj_exact> <pred_obj> <pred_phrase_exact>."

"<preamble_almost> ::= <subject_almost>
<subj_modifier_almost> <prep_phrase_almost>
<prep_phrase_1_almost> <prep_phrase_2>
<prep_phrase_3_almost> <prep_phrase_4>
<prep_phrase_5> <prep_phrase_6a>
<prep_phrase_6b_almost> <verb_phrase_almost>
<pred_adj_almost> <pred_obj>
<pred_phrase_almost>."
```

Tips for Forming Grammars

The following tips may be useful to keep in mind when forming BNF grammars:

- A grammar may fail to work properly if any terminal is misspelled, so use a spell check to find errors.
- Use standard spelling for your grammar; don’t try to spell words phonetically.
- A grammar may fail to compile properly if a rule is undefined. If the grammar doesn’t work as it was intended, check the BNF syntax.

Example Grammars

Example grammars are a good place to observe best practices for creating BNF grammars in applications created using the SDK.

This example demonstrates a complex grammar that is listening for multiple commands.

```
"<command> ::= <schedule> | <next>."
"<schedule> ::= TODAYS | SCHEDULE |
TODAYS_SCHEDULE."
"<next> ::= <adj> <subject> | <subject>."
"<adj> ::= JUST | JUST_MY."
"<subject> ::= NEXT | NEXT_MEETING."
```

This example below demonstrates a way to create a recursive grammar that listens for noise by using the noise phone symbol ($).

```
"<any> ::= <noise> <any> | <noise>."
"<noise> ::= $AA | $AE | $AO | $AX | $AXR | $B |
$BD | $DD | $EH | $EY | $K | $L | $M | $IY | $N
| $R | $SH | $T | $TD | $V | $Z."
```

The example below demonstrates a complex grammar that is listening for multiple phrases and synonyms. It also references the above rule for `<any>`, to account for noise that may interfere with the speech recognition.

```
"<phrase> ::= <verb> <subject> | WHAT_CAN_I_SAY."
"<verb> ::= <any> | <any> GET_MY | GET_MY."
"<subject> ::= <mail> | <calendar> | <contacts>."
"<mail> ::= MAIL | EMAIL."
"<calendar> ::= CALENDAR | SCHEDULE."
"<contacts> ::= CONTACTS | ADDRESS_BOOK."
```

The above grammar could potentially even recognize a sentence preceded by a user who coughed before speaking.

The following example demonstrates a grammar that is listening for an adjective and an optional noun.

```
"<email> ::= <adj> | <adj> MAIL."
"<adj> ::= ALL | UNREAD | JUST_UNREAD."
```

The following examples demonstrate simple lists of things the user can say.

```
"<calendar_flow> ::= STOP | STOP_READING | NEXT |
```

```
NEXT_MEETING | SKIP | GO_BACK."
"<contact> ::= MARK_ADAMS | LINDA_BAKER |
THOMAS_CHAVEZ | GEORGE_HILL | PATRICIA_ROBERTS |
MARY_RUIZ | STEPHANIE_SMITH | DAVID_THOMPSON |
AMY_WANG | BRIAN_WHITE."

"<contact_flow> ::= ADDRESS | PHONE_NUMBER | MAIL |
EMAIL | ALL_INFORMATION |
ALL_CONTACT_INFORMATION."

<email_flow> ::= STOP | STOP_READING | NEXT |
NEXT_ITEM | NEXT_MESSAGE | SKIP | GO_BACK."
```

4 Glossary

barge-in
The ability to interrupt audio output as a result of recognition of a user utterance. Barge-in is a mode of the application and is only possible in a full-duplex system.

BNF (Backus-Naur Form)
A notation for describing the syntax of a language. The grammar format that is recognized by CASSI™. For example, the notation for a grammar rule called `<filemenu>` could be written like this:

```
<filemenu> ::= OPEN_FILE | CLOSE_FILE
```

CASSI (Conversay Advanced Symbolic Speech Interpreter)
The core speech recognizer and synthesizer used by Mobile Conversay™ SDK.

CASSI Services
The API that interacts with CASSI. May be accessed through C or C++.

class factory
An object that facilitates the creation of instances of the root object. The class factory of the CASSI Services API is `CVACreateInstance`.

context
A logical grouping of topics. For example, in a speech application that provides access to flight reservations, topics that are related to a particular itinerary may be grouped into one context. Ticket price and seat preferences would be two other contexts.

conversational focus
The context and topics that are active. Referred to as “in conversational focus.” Inactive topics are frequently referred to as “out of conversational focus.”

duplex
A measure of an audio system’s capability to handle sound input and output. A full-duplex system is capable of simultaneous input and output, allowing for barge-in or for recording and playing sound simultaneously. A half-duplex system must alternate between input and output.

embedded Linux®
The Linux-based platform for mobile devices, including PDAs and hand-held computers.

grammar
A set of language rules that aids recognition accuracy and response time in speech recognition systems by constraining what the speech engine listens for. For example, a grammar may contain a rule called “filemenu” that recognizes only “open file” and “close file.” BNF (Backus-Naur Form) is the grammar format recognized by the CASSI speech engine.

noise phones
Noises that do not coincide with the phoneme set of a language.

noise phone level
The rate at which the system recognizes noise phones, relative to normal recognizable speech.

PDA (Personal Digital Assistant)
A small mobile hand-held device that provides computing capabilities for personal or business use. PDAs are typically used for keeping address book and schedule information, in addition to meeting other mobile computing needs.

phoneme set
The abstract units of a language’s phonetic system. These units correspond to a set of similar speech sounds, which are perceived to be a single distinctive sound by human listeners.

Pocket PC
The Microsoft® Windows®-powered platform for PDAs and hand-held computing.

prompt
In a speech application, an audible or visual cue that indicates that it is the user's turn to speak.

reco (speech recognition)
The ability to take a voice waveform and match it to a specified grammar.

ref-count (reference-count)
A count that tracks accessing and closing a topic in a grammar. When a topic in a grammar is accessed or closed, its ref-count number is incremented or decremented. Topics with a ref-count number of *0* are deleted to free system resources.

STP (spelling-to-pronunciation)
A module that allows the CASSI speech engine to synthesize and recognize speech without looking up the pronunciation in the speech engine's dictionary module. Without STP rules, applications must have all grammar items loaded in a dictionary to allow the grammar to be compiled.

threshold (or voice threshold)
An amplitude energy level that must be overcome before a waveform will be sent to the recognition engine.

topic
Specification of the words that can be recognized by the speech engine in a particular context. For example, in a speech application that provides banking services, topics in the checking account context could specify words that are related to transaction dates, check numbers, payments, and deposits.

TTS (text-to-speech)
The synthesis of text into speech waveforms. The text-to-speech capability of the CASSI speech engine includes text normalization and prosody processing.

utterance
A single spoken event. May consist of a single word or of several words spoken continuously.

VUI (voice user interface)
A user interface that includes speech recognition, recorded speech output, and synthetic speech output to communicate with the user.

WCIS (What Can I Say?)
A help service that informs the end user of the commands that the system is listening for. For example, a WCIS response could be, “You can say coffee, tea, milk, or no beverage.”

WYS (What You Said)
A help service that informs the end user of what the speech engine has recognized as input.

Index

Index

Index

Index

Index

Index