CT Application Development Environment

Getting Started Guide

Software/Version: Envox Computer Telephony Application Development Environment (CT ADE) 9.2

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Envox Group AB.

Envox, Envox CT Connect and the Envox logo are registered trademarks or trademarks of Envox Group AB or its affiliates in the United States and other countries. All other trademarks and registered trademarks referenced herein are the property of their respective owners.
Copyright © 2007 Envox Group AB. All rights reserved.

Envox CT Application Development Environment Getting Started Guide
Edition: November, 2006

For Sales Offices and other contact information, visit our web site at http://www.envox.com

Contents

Preface 7

1 CT ADE Overview

1.1 CT ADE Overview 1-1
1.2 The Resource Manager 1-3
1.3 Evaluation and Development Modes 1-3
1.3.1 Evaluation Mode 1-3
1.3.2 Development Mode 1-3
1.4 Application Development Environments 1-4
1.4.1 AD Language 1-4
1.4.1.1 ADL Language Compiler and Runtime Engine 1-4
1.4.1.2 Editor 1-4
1.4.1.3 Runtime Link Libraries 1-5
1.4.1.4 Debugger 1-5
1.4.2 AD ActiveX Objects 1-5
1.4.2.1 ADX Components 1-5
1.5 Additional Tools and Utilities 1-6
1.5.1 Simulated Phone 1-6
1.5.2 Sample Applications 1-6
1.5.2.1 ADX Samples 1-6
1.5.2.2 ADL Samples 1-6
1.5.3 Media Toolbox 1-7
1.5.4 Audiotst 1-7
1.5.5 WaveTest 1-7
1.5.6 Hardware Key 1-8

2 Inside the Resource Manager

2.1 Resource Manager Overview 2-1
2.1.1 Resource Manager Resources 2-3
2.1.1.1 Trunk Resources 2-5
2.1.1.2 Media Resources 2-5
2.1.1.3 Fax Resources 2-5
2.1.1.4 Voice Recognition Resources 2-5
2.1.1.5 Text-To-Speech Resources 2-6
2.1.1.6 Conferencing Resources 2-6
2.1.2 Resource States 2-6
2.2 The Resource Manager Profile 2-8
2.2.1 Resource Manager Profile IDs 2-9
2.2.2 Resource Manager RegIDs 2-10
2.2.3 TZP Files 2-10

2.2.3.1 IPFs.tzp File 2-10
2.2.4 Running the Resource Manager Scanner 2-12
2.3 CTADE.ini File 2-13
2.4 Languages in the Resource Manager 2-14
2.4.1 Language Files 2-14
2.4.1.1 IPF 2-15
2.4.1.2 Language Profile 2-15
2.4.1.3 TZP Files 2-15
2.4.1.4 Languages.tzp File 2-15
2.4.1.5 Text Files 2-16
2.4.2 Phrase Element Parameters 2-16
2.4.2.1 Value Parameter 2-16
2.4.2.2 Type Parameter 2-17
2.4.2.3 Gender Parameter 2-17
2.4.2.4 Language Parameter 2-18

3 Using Simulated Phone

3.1 Overview 3-1
3.2 Starting Simulated Phone 3-1
3.3 Incoming Calls 3-2
3.4 Phone Line Properties 3-3
3.5 Outbound Calls 3-3

4 Getting Started with ADX

4.1 Overview 4-1
4.1.1 ADX Components 4-1
4.1.1.1 ADX Voice Object 4-2
4.1.1.2 ADX Fax Object 4-2
4.1.1.3 ADX Conference Object 4-2
4.1.1.4 ADX TTS Object 4-2
4.1.1.5 ADX ASR Object 4-2
4.1.1.6 Media Toolbox 4-3
4.1.2 ADX Programming 4-3
4.1.2.1 Methods 4-3
4.1.2.2 Properties 4-3
4.1.2.3 Events 4-3

5 ADL Concepts

5.1 Overview 5-1
5.1.1 Source Code Basics 5-1

5.1.2 Source Code Example 5-2
5.2 Writing Simple Expressions 5-2
5.2.1 Values 5-2
5.2.1.1 Numerical Values 5-3
5.2.1.2 Logical Values 5-3
5.2.2 Variables 5-3
5.2.2.1 Example Program with a Variable Declaration Block 5-4
5.2.3 Constants 5-4
5.3 Forming Complex Expressions 5-4
5.3.1 Arithmetic Operators 5-5
5.3.1.1 Arithmetic Using Decimal Points 5-5
5.3.2 Logical Operators 5-6
5.3.3 Comparison Operators 5-7
5.3.3.1 Assignments 5-7
5.3.3.2 String Concatenation 5-8
5.4 Using Statements 5-9
5.4.1 Switch/Case Statements 5-9
5.4.2 Goto Statements 5-10
5.4.3 Jump Statements 5-11
5.4.4 Include Statements 5-11
5.5 Constructing Loops 5-12
5.5.1 For Loop 5-12
5.5.2 While Loop 5-13
5.5.3 Do...Until Loop 5-13
5.5.4 Using loops 5-13
5.6 Utilizing Functions 5-14
5.6.1 Built-In Functions 5-15
5.6.2 User-Defined Functions 5-16
5.6.3 Function Files and Libraries 5-18
5.6.4 RLL Functions 5-18

6 Writing ADL Script

6.1 Starting the ADL Program Tutorial 6-1
6.2 Creating a Project for Your Program 6-1
6.3 Adding a Program to the Project 6-2
6.4 Compiling and Running the First Program 6-3
6.4.1 Reading the Source Code 6-4
6.4.2 Creating a Second Call Processing Program 6-4
6.4.3 Compiling and Running the Second Program 6-5
6.5 Examining the Second Program 6-6
6.5.1 Echo Program Code Detail 6-6
6.6 Using Basic Telephony Functions 6-8

6.6.1 Audio Processing 6-8
6.6.2 Telephone Signaling 6-8
6.6.3 Programming Basic Features 6-8
6.6.3.1 Answering an Incoming Call 6-9
6.6.3.2 Making an Outgoing Call 6-9
6.6.3.3 Terminating a Call 6-10
6.6.3.4 Detecting a Disconnect 6-10
6.6.3.5 Creating a Touch Tone Menu 6-11

7 Debugging in ADL

7.1 Creating an Application 7-1
7.1.1 Before You Start 7-1
7.1.2 Creating a New Project 7-1
7.2 Debugging the Application 7-3
7.2.1 Starting Debugger 7-3
7.2.2 Stepping Through Code 7-4
7.2.3 Using Breakpoints 7-5
7.2.4 Viewing the Call Stack 7-5

8 Troubleshooting

8.1 Overview 8-1
8.2 Troubleshooting in ADL 8-2
8.2.1 Viewing the Log File 8-2
8.2.2 Interpreting the ADL.LOG Contents 8-3
8.2.2.1 ADL Version 8-3
8.2.2.2 ADL Exit 8-3
8.2.2.3 Using Built-In Function Calls 8-3
8.2.2.4 Blocking Function Calls 8-4
8.2.2.5 Blocking Functions and Resource Manager State Changes 8-4
8.2.2.6 Example: Recognizing Blocking in Log Entries 8-4
8.2.2.7 Asynchronous Mode 8-5
8.2.2.8 Example: Recognizing Blocking in Log Entries in Asynchronous Mode
8-5
8.2.2.9 Interpreting @ Codes in the Log File 8-5
8.2.3 Configuring the ADL.LOG File 8-6
8.2.3.1 Configuring the Logging Dialog 8-7
8.2.3.2 Configuring the Tracing Dialog 8-9
8.3 Troubleshooting in ADX 8-10
8.3.1 Interpreting the ADX.log File Contents 8-11
8.3.2 Controlling Log Output 8-13

9 Technical Support

9.1 Overview... 9-1
9.2 What Support Needs From You 9-2

A Glossary

Index

Preface

This chapter describes how this book is organized and contains the following sections:

- How to Use this Guide
- Documentation Suite
- Hardware and Software Requirements
- Contacting Us

How to Use This Guide

This section explains what you need to know before using Envox™ Computer Telephony Application Development Environment (CT ADE) and describes the content and layout of this guide.

Required Knowledge

To use Envox CT ADE, you need to be familiar with:

- Windows® 2000 or later operating systems
- Computer telephony terminology and concepts
- Computer telephony hardware
- If you plan to use AD ActiveX Objects (ADX), you need to know one of these programming languages: Visual Basic .NET, Visual C++.NET, Visual C# .NET, or Delphi.

Format Conventions

The following conventions should help you navigate more easily through this guide.

Displaying Code Examples

Text in Courier font indicates computer code, for example:

```
program
    vid_write("I am an ADL program!");
    vid_write("Type any key to exit...");
    kb_get();
    vid_write("Exiting now.");
    exit(0);
endprogram
```

Navigating Application Menus

When describing which submenu item to choose from a higher-level application menu, the pipe symbol (|) is used. For example: From the Start menu, choose Programs | Envox CT ADE | ADX | ADX Help.

Introducing New Terms and Showing Audible Messages

Text displayed in italicized style is used to introduce a new word or concept. Subsequent use of the term appears in normal text.

Italicized text is also used to demonstrate that a Computer Telephony application is playing an audible message. For example:

"Press 1 to hear new messages, press 2 to hear saved messages, press 3 to..."

Identifying Text That You Need to Type

Text that you need to type usually shows up in a sequence of steps and specifies exactly what you need to type in order to complete the step. It displays in either bold style or Courier font.

For example, in the following sentence you type the word "inbound" in the field specified or on the command-line described:

1. Name the file **inbound**, and make sure the Add to Project box is checked.

   Code that you need to type can also display in Courier font. The following step shows an example of this:

2. Click in the Editor window on the right and type the following code into it:

```
program
     TrunkWaitCall();
     TrunkAnswerCall();
     if (TrunkGetState() strneq "Connected")
         voslog("Unexpected trunk state ", TrunkGetState());
         stop;
     endif
     InboundCall();
     TrunkDisconnect();
```

```
    restart;
endprogram
```

Documentation Suite

CT ADE comes with a variety of online documentation to help you use the product.

Online Help Files	Portable Document Format
ADL User's Guide	CT ADE Getting Started Guide (this guide)
CT ADE Profile Help	
ADX User's Guide	
Simulated Phone Help	
Sample Application ReadMes	
WaveTest Utility Help	
Audiotst.exe ReadMe	
Release Notes	

Online Help Files

When you install CT ADE, menu items are created on the Start menu that take you to the latest online help files. We strongly recommend that you spend some time exploring the online help to see what information is available. Investing a few hours to do this will repay itself many times over when you begin developing CT systems.

ADL User's Guide

The ADL online help file describes what the ADL development environment is and how to set it up for use. It also describes what the Resource Manager is and how it works with ADL. You can find extensive information about using an ADL script, including tutorials to help you begin developing your applications. There is also a tutorial on using the Debugger tool.

The online help file also includes tables that display the Resource Manager RegIDs for Get/Set functions (RegIDs values are used internally by the Resource Manager) so you can retrieve information about the system or issue a Technology-specific command at runtime.

The help file contains information about the Runtime Link Libraries (RLLs) available with ADL that you can use to add additional functionality to your applications.

- ADO RLL—Provides ADL applications with direct access to Microsoft ActiveX Data Objects (ADO).
- Fixed Point Math RLL—Provides functions to add, subtract, multiply, and divide

decimal numbers from within ADL programs.

- Inter-Chassis Routing RLL—Supports integration between CT ADE and the Amtelco H.100 MC3 Multi-Chassis Interconnect board.
- Network Hub RLL—Provides communication between ADL tasks and/or ActiveX applications.
- Web RLL—Has functions that enable your ADL program to: get documents from a Web server, access data from active Web pages (for example, from an e-commerce server), and send and receive files via FTP.

Finally, you can find information on logging and tracing, troubleshooting problems, and contacting Technical Support.

ADX User's Guide

The AD ActiveX Objects (ADX) online help file tells you how to set up the ADX development environment and describes the functions of the various components contained in ADX. Also described are the methods, properties, and events belonging to each component and details about how to use them. Additionally, you will find information about the Resource Manager and how it works with ADX.

Simulated Phone Help

The Simulated Phone online help file describes how to use the Simulated Phone and includes information about compatibility with ADX and ADL.

Sample Application ReadMes

The sample applications are provided to give you a few examples of the kinds of applications you can write and you are encouraged to copy and modify these samples for your own use. Each sample has a ReadMe that describes what the application does and how it. If you are using ADL, you can view Script samples (samples written in the ADL programming language). If you are using ADX, you can view sample applications in Delphi, Visual C++, and Visual Basic. Sample applications are included in the default installation and you can access them from the Start menu.

WAVETEST Utility Help

The WAVETEST online help file accompanies WAVETEST, which is a command-line utility that lets you manage Wave devices and files. The help file describes how to display installed Wave devices, test your system for supported Wave formats, and display the a Wave file's format. This file also includes information to help you troubleshoot problems.

Audiotst.exe ReadMe

The Audiotst.exe ReadMe describes how to use this application to find out the number of voice boards present on your system.

License Upgrade Help

The License Upgrade online help explains how to use the License Upgrade utility to upgrade your existing license.

CT ADE Profile Help

The CT ADE Profile online help provides an overview of the profile and how it is used by CT ADE. It also describes the Profile Configuration Utility.

Portable Document Format Documents

CT ADE Getting Started Guide

The Getting Started Guide (this guide), describes the two development environments and various components that make up CT ADE. The Portable Document Format (PDF) version of this guide is installed by default during installation of CT ADE.

Hardware and Software Requirements

This section gives you a general overview of hardware and software requirements you need to run CT ADE. However, depending on whether you are running CT ADE in evaluation mode or development mode, and what hardware, drivers, and speech engines you have installed, etc., your requirements will vary.

Hardware Requirements

To run CT ADE, you need the following hardware:

- Telephony Hardware and Software

  Refer to the documentation provided with your Intel® Dialogic® board(s) for information on hardware installation. For information board drivers, see your Intel Dialogic distributor.

  You can also use Intel® NetStructure® Host Media Processing Software in place of hardware boards.

  If you are running in evaluation mode, you may not need telephony boards or drivers; Simulated Phone can be used for development and testing of many runtime applications.

- Hardware Key

  If you plan to run CT ADE in development mode, a hardware key needs to be installed on your PC. This key, which may be included with your installation materials, is programmed with information about your license. A hardware key can connect to a standard parallel or USB port. Parallel port hardware keys allow normal operation of printers and other devices that can be attached to the port.

  You do not need the hardware key if you are running CT ADE in evaluation mode.

- How many lines can my PC Run?

  A commonly asked question is how much hardware (processor speed, RAM, etc.) is required to run a given number of telephone lines. The answer depends on many factors relating to your particular application. Your distributor can advise you about specific information for your particular configuration.

  You do not need any phone lines if you are running in evaluation mode—you can use Simulated Phone instead (unless you are evaluating Text-To-Speech, Voice Recognition, Conferencing, or Faxing capabilities; in this case you'll need the appropriate board/speech engine). For more information about Simulated Phone, see Chapter 3.

Software Requirements

To run CT ADE, you need the following software:

- PC running Windows 2000 or later operating system
- The Resource Manager, which is automatically included in the CT ADE installation

  For information about configuring the Resource Manager, see the online help.

- Simulated Phone, which is automatically included in your CT ADE installation

  You only need Simulated Phone if you do not plan to use any telephony hardware.

- To use ADX, you need access to a programming language you are familiar with, such as Visual Basic .NET, Visual C++ .NET, Visual C# .NET, or Delphi.
- Third-party applications

  To develop certain applications that use third-party products and to run certain sample applications, you need additional third-party software such as Nuance Text-To-Speech software. The sample application ReadMes, ADL online help, or ADX online help list the requirements in these cases.

1

CT ADE Overview

This chapter describes the Envox CT ADE™ suite of products:

- CT ADE Overview
- Application Development Resource Manager
- Application Development Environments
- Additional Tools and Utilities

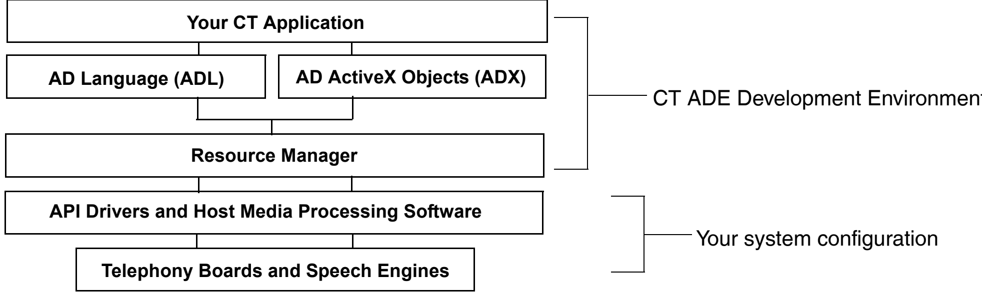
1.1 CT ADE Overview

Computer Telephony Integration (CTI) is an implementation of telephone and computer technology that enables voice and data processing equipment to work together, letting you efficiently and easily exchange information via tools such as telephones, computers, faxes, and voice recorders.

We've utilized this technology to develop CT ADE, which stands for computer telephony (CT) application development environment (ADE). CT ADE is made up of the following components: the AD Resource Manager engine (Resource Manager), the AD Language (ADL) programming environment, and the AD ActiveX Objects (ADX) programming environment. Together, these components provide an environment in which you can develop, test, debug, and run your own CT applications. CT ADE also communicates with and utilizes the API drivers, telephony boards, and any additional speech engines already installed on your system to perform CT functions.

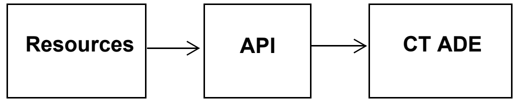
CT ADE offers development tools and a choice of two programming interfaces, ADX or ADL, that eliminate the need to write directly to the API in C or C++. Additionally, the underlying Resource Manager architecture supplies an abstraction layer that sits on top of an API and executes low-level CT functions so you don't have to worry about differing hardware and API protocols. The Resource Manager commands you use to initiate functions are simple to use, easy to learn, and consistent.

Figure 1–1 CT ADE development environment and system configuration correlation



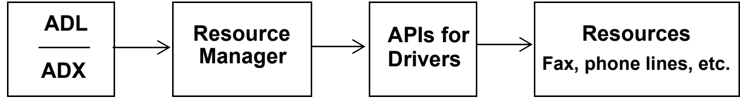
Before installing CT ADE, you must first add any phone lines, fax machines, etc., that you plan to use in your telephony application. Next, you install the APIs for the drivers, and finally you install CT ADE.

Figure 1–2 Direction flow of system installation



Following installation, you run a scanner utility that creates a database and populates it with configuration information about any telephony boards and speech engines installed on your system. At this point you can begin developing your CT application, which references the database via the Resource Manager to carry out the CT functions you request using commands called functions in ADL and methods in ADX.

Figure 1–3 Direction of application flow



The commands are easy to use regardless of trunk type or API. For example, if you invoke the ADL function *TrunkAnswerCall,* the Resource Manager layer determines which trunk interface is being used and which API function to use: *dx_sethook* (analog SpringWare trunk) or *gc_AnswerCall* (Global Call trunk). We are committed to keeping Resource Manager applications portable to different APIs and trunk types as the industry evolves and customer needs change.

1.2 The Resource Manager

The design of the Resource Manager builds on more than a decade of experience in dealing with telephony APIs, cross-platform portability, and the wide variation in capabilities offered by telecommunications hardware and services. The Resource Manager mediates between your programming commands and any telephony boards and speech engines installed on your system in order to carry out the commands your program executes.

The most important feature of the Resource Manager, however, is API Transparency. The Resource Manager relieves you from dealing with many chores that native APIs require, such as complex API function signatures including handles, bit-fields, pointers and structures. The Resource Manager also determines the installed hardware and trunk configuration. Additionally, the same set of functions and methods work under all supported telephony APIs (Intel Dialogic R4 or greater, Intel Dialogic Global Call, etc.) and all supported trunk types (analog loop-start, T-1 E&M, E-1/R2, PRI, etc.). Compared with C programming, this means that typical application functionality such as making or receiving calls, or playing menus and prompts, is substantially easier to create and maintain.

Finally, the API transparency the Resource Manager provides lets you avoid locking yourself into committing to a particular API. Unlike a C/C++ program, your Resource Manager application supports portability to different APIs and trunk types as the industry evolves and your end-user needs change.

1.3 Evaluation and Development Modes

After installing CT ADE, you choose an ADE (ADX or ADL) that you run in either evaluation mode or development mode. With a valid hardware key attached to your PC, the software runs in full development or runtime mode. Without the hardware key, the evaluation mode runs.

1.3.1 Evaluation Mode

The following restrictions apply when you run in evaluation mode:

- Your applications cannot control telephony boards so you must use Simulated Phone to simulate making or receiving a phone call.
- Applications that were compiled in Evaluation mode must always use Simulated Phone, even if a runtime hardware key is later attached to the PC.

1.3.2 Development Mode

When you attach a valid hardware key to your PC, ADL and ADX can simultaneously run multiple instances of each of these Resource types: Trunk Resources, Media Resources, Fax

Resources, Conference Resources, Text-To-Speech Resources, and Voice Recognition Resources.

Applications compiled in development mode run on as many API/driver combinations (called Resources) that the attached development or runtime key will allow. For example, if you wanted to run an application with 64 Media Resources, 30 Conference Resources, and 64 Fax Resources, you would need to attach a 64-port hardware key to the PC.

For more information about Resources or the hardware key and how to configure it, see either the ADL or ADX online help.

1.4 Application Development Environments

CT ADE offers you a choice of two environments in which to develop your applications: ADL or ADX. Each environment has been designed with unique advantages and one is sure to suit your needs.

1.4.1 AD Language

With the AD Language (ADL), you can write and maintain standalone computer telephony (CT) applications. Similar looking to C, ADL was specifically designed to support CT technology. The coding process is simpler than coding in C or C++ because much of the application details are handled by the ADL engine, which also make maintenance and modifications easier to manage.

You can create and deploy telephony systems with a complete development environment that includes the Resource Manager, the ADL Language Compiler and runtime engine, Editor, Runtime Link Libraries, Debugger, and Simulated Phone.

For information about the Simulated Phone, see Section 1.5.1.

1.4.1.1 ADL Language Compiler and Runtime Engine

ADL Compiler translates ADL source code into an ADL executable, which can then be used by the ADL runtime engine.

ADL provides an environment in which you can write, test, debug, and run your CT applications in a Windows environment. The ADL engine is responsible for executing the applications built with the ADL Compiler.

1.4.1.2 Editor

The ADL Editor is a syntax-highlighting editor designed specifically for ADL language source code. As you edit code, the Editor highlights and colorizes different ADL elements, making it easy to follow the flow of your code. You can use the Editor to create new ADL source and function files, or to edit existing ones.

1.4.1.3 Runtime Link Libraries

Runtime Link Library (RLL) add-ins are extensions that expand ADL functionality. ADL comes with a number of RLLs implemented as Dynamic Link Libraries (DLLs) that let you load executable code modules on demand to be linked at runtime. RLL functions can be called from an ADL application like other ADL functions.

For example, the Network Hub RLL lets you send and receive messages to ADL tasks running on remote PCs that are connected via an IP network.

1.4.1.4 Debugger

The Debugger is a source-code level graphical debugger that lets you test and troubleshoot ADL programs through the development phase and after installation and deployment. You can use this tool to debug many programs in a single session and to check program variables and the current state of your application without interrupting the calls in progress.

To help you become more familiar with the Debugger, this guide includes a tutorial in which you create an application in ADL and then use the Debugger to debug it. For more information, see the ADL online help.

> **Note:** The Debugger is not for use with CT applications developed in the ADX environment.

1.4.2 AD ActiveX Objects

AD ActiveX Objects (ADX) are an alternative set of tools that you can use to add telephony features to legacy applications or to create new applications from scratch.

ADX is made up of the Resource Manager, Simulated Phone, a collection of ActiveX components, and any Windows development environment that supports ActiveX components. The most commonly used programming language is Visual Basic 6, and others including Visual C++, C#, and Visual Basic .NET.

For details about Simulated Phone, see Section 1.5.1.

1.4.2.1 ADX Components

The ADX ActiveX components add the following capabilities to your CT applications:

- ADX Voice Object component —Handles call control, playing and recording of sound files and tones, and getting DTMF (Dual Tone Multi-Frequency) digits from callers
- ADX Fax Object component—Lets you add fax support to your applications
- ADX Conference Object component—Adds conferencing and call center switching capabilities to your applications
- ADX TTS Object component—Controls text-to-speech features in your applications

- ADX ASR Object component—Enables your application to recognize speech
- ADX ICR—Inter-Chassis Routing
- Network Hub—Inter-Process Communications

1.5 Additional Tools and Utilities

Various additional tools, utilities, and sample applications are automatically included in the CT ADE installation and are available from the Start menu following installation. Depending on your purchase, the hardware key may also be included in your CT ADE package.

1.5.1 Simulated Phone

Simulated Phone is a software phone line simulator that works with either ADX or ADL and the Resource Manager to simulate a telephony board and phone line. The simulation is accomplished using the screen, keyboard, mouse, and sound card installed in your PC. Simulated Phone can be used whether or not a telephony board is installed.

1.5.2 Sample Applications

Use the Start menu to access the sample applications for viewing or to modify them for use with your own applications.

1.5.2.1 ADX Samples

ADX has sample applications written in Visual Basic .NET, Visual C#, and Delphi. The Visual Basic samples include applications that allow callers to do their banking by phone, send and receive faxes, and monitor both inbound and outbound calls. Additional applications use CT ADE Network Hub communication technologies. There are also several Voice Recognition and Text-To-Speech applications that utilize third-party software. Both the Visual C# and the Delphi samples provide a multithreaded inbound call application.

1.5.2.2 ADL Samples

ADL has sample applications available in ADL script, written using ADL Studio.

The Script samples include numerous applications that let callers access movie title and showtime information, perform database operations, and monitor inbound and outbound calls. Additional samples incorporate technologies such as Text-To-Speech, AD Communications Hub communication, and Voice Recognition using third-party software.

For information about Text-To-Speech, Voice Recognition, and multithreaded applications, refer to the ADL or ADX online help.

1.5.3 Media Toolbox

Media Toolbox is a family of ActiveX controls that further help you modify your CT applications. It contains the following:

- Sound File Converter ActiveX and DLL for converting between Vox and Wave file formats
- IPFile ActiveX for creating and editing Indexed Prompt Files (also known as VAP, macro, or VBase/40 files)
- WaveDevice ActiveX for querying the capabilities of your installed Wave devices
- WaveFile ActiveX for querying the format of a Wave file

There are also sample applications that you can use with your own sound files.

The Media Tools executables are located in the following default installation directory:

```
C:\Program Files\Envox\CT ADE\Media Tools
```

The samples are located in:

```
C:\Program Files\Envox\CT ADE\Media Tools\Samples
```

The online help for these tools is located in the *ADL User's Guide* and *ADX User's Guide*.

1.5.4 Audiotst

Audiotst is an application that determines the number of voice boards present on your system and then plays a set of Wave files in various formats to each board to find out if those formats are supported. If they aren't, it reports an error message. You can use Audiotst with both ADL and ADX.

You can find the Audiotst executable and ReadMe file in the following default installation directory:

```
C:\Program Files\Envox\CT ADE\Common\Bin
```

1.5.5 WaveTest

WaveTest is a command-line utility that allows you to check your system for installed wave devices and to find out what their capabilities are. You can use WaveTest with both ADL and ADX.

You can find the WaveText executable in the following default installation directory:

```
C:\Program Files\Envox\CT ADE\Common\Bin
```

You can find the WaveText online help file in the following default installation directory:

```
C:\Program Files\Envox\CT ADE\Help
```

1.5.6 Hardware Key

The hardware key, also called a dongle, must be attached to a parallel or USB port on your PC in order for CT ADE programs to function correctly. It has a standard parallel or USB port connector and allows normal operation of printers and other devices that can be attached to the port. If you are running in evaluation mode the key is not required.

Caution: We recommend removing the hardware key from the port when not needed, especially when using bi-directional software such as At Work printers, LapLink and other file transfer utilities. Occasionally, some types of hardware devices can be damaged by use of bi-directional software or by static electricity.

2

Inside the Resource Manager

This chapter describes the features of the Resource Manager and how it works. It contains the following sections:

- Resource Manager Overview
- Resource Manager Profile
- CTADE.ini File
- Languages in the Resource Manager

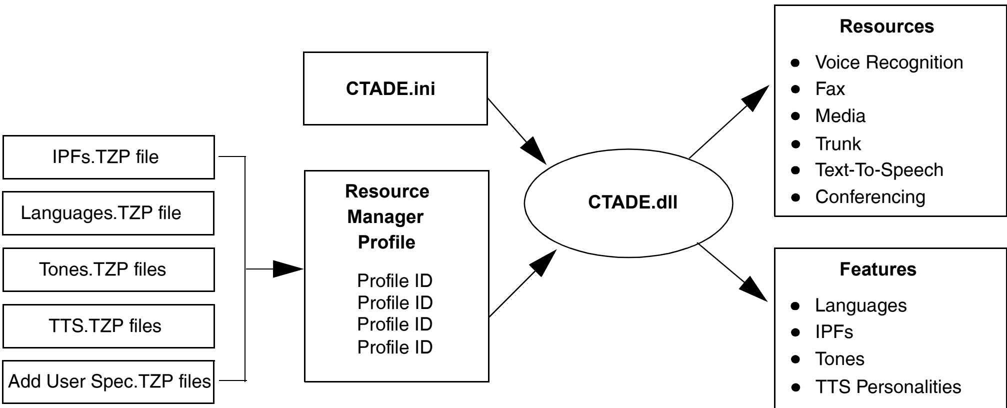
2.1 Resource Manager Overview

The Resource Manager portion of Envox CT ADE™ is a set of software components that handles the CT technology functions in your applications. The main components of the Resource Manager are the CTADE.dll, the Scanner, the Profile, and the CTADE.ini file. The Resource Manager also includes a collection of languages that you can use with your applications.

The Resource Manager Scanner is a utility that you run to gather information about speech engines and telephony boards installed on your system. It then builds a database called the Resource Manager Profile in which to store this information. Finally, the CTADE.ini file is used to specify configuration parameters for the Resource Manager and Resource Manager Scanner.

The Resource Manager also stores a set of languages and administers information about them for use with Phrase functions. For example, if you want to speak a phrase in Italian, the Resource Manager tracks where the Italian language prompts are located and uses the pre-defined rules included in the Italian directory to use the prompts correctly.

Figure 2–1 Flow of Information: Resource Manager Components



The Resource Manager mediates between programming commands such as playing a file and the underlying hardware and API combination. While the various components of the Resource Manager are referenced while your application is running, the object-oriented architecture of the Resource Manager is structured so that only CTADE.dll operates at runtime and the performance of your applications is not compromised. The telephony boards and speech engines as well as their configuration and API combinations can be separated into groups called Technologies. The Resource Manager gathers this Technology information from the Resource Manager Profile, translates these into usable Resources, and then uses them to carry out the CT commands of your application.

For example, if your program invokes the ADL function TrunkAnswerCall, the Resource Manager engine determines which trunk interface is being used, whether it is legal to make this call, and chooses which API function to use, such as *dx_sethook* or *gc_AnswerCall*.

Technologies are named using the convention <API><Resource Type> For example:

- R4DxMedia is a media resource based on Intel Dialogic R4 dx_ API.
- R4AgTrunk is an Analog trunk interface based on Intel Dialogic dx_/ag_ API. LSI interfaces found on boards such as Proline/2V, Dialog/4, D/41ESC, VFX/40, and D/160SC-LS use these APIs.

There are many Technologies but each falls into one of the six Resource types as follows:

Resource Manager Technology	Description	Resource Type
XXXTrunk	Responsible for all call control, including dialing out, accepting an incoming call, and hanging up when a call is finished	Trunk
XXXMedia	Controls all playing and recording of sound files and tones, as well as getting DTMF digits from callers	Media
XXXFax	Controls the transmission and processing of fax data	Fax
XXXVR	Controls voice recognition	VR
XXXTTS	Controls text-to-speech	TTS
XXXConf	Controls individual conferences	Conference

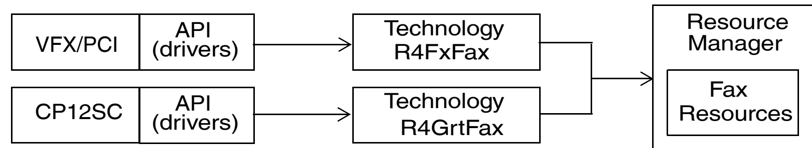
For a complete list of the Technologies currently available, refer to the ADL or ADX online help.

2.1.1 Resource Manager Resources

A Resource is a component of a call processing system. In most cases, a single Resource processes a single stream of audio corresponding to the sound on a telephone line or channel on a digital trunk. Technologies get interpreted into Resources by the Resource Manager.

Figure 2–2 shows how the Resource Manager gathers hardware and API configuration information and translates it into a resource.

Figure 2–2 Resource Manager Resource Process



Resources are identified when you run the Resource Manager Scanner and information about them gets stored in the Resource Manager Profile. Resources are managed by the Resource

Manager, and controlled by functions in ADL and methods or functions in ADX. Each Resource produces and/or consumes a single stream of audio, as follows:

Resource Type	ADL Function	ADX Component Method
Trunk	TrunkXXX functions	ADX Voice Object methods
Media	MediaXXX functions	ADX Voice Object methods
Fax	FaxXXX functions	ADX Fax Object methods
VR	VrXXX functions	ADX ASR Object methods
TTS	TtsXXX functions	ADX TTS Object methods
Conference	ConfXXX functions	ADX Conference Object methods

When ADL or ADX and the Resource Manager are started, each Resource is assigned a Resource index number, starting from 0 to the total number of that type of Resource on your system minus one. Generally, you won't have to worry about Resource index numbers, since ADL and ADX reserve and route Resources as your application needs them, but if you want to control how your applications use Resources, you can specify them by their index number. Index numbers start at 0 for each Resource type, and are numbered independent of other Resource types, so an ADL task could easily be using Trunk Resource 1 and Media Resource 4.

Individual Resources of a given Technology type are identified with Resource Index Numbers. So, for example, if your system had a D/240SC-T1, a D/41ESC, and a VFX/40ESC telephony board, you would have the following Resources under the Resource Manager:

D/240SC-T1:	24 Trunk Resources	24 Media Resources	
D/41ESC:	4 Trunk Resources	4 Media Resources	
VFX/40ESC:	4 Trunk Resources	4 Media Resources	4 Fax Resources
Totals:	32 Trunk Resources	32 Media Resources	4 Fax Resources

This system's Trunk Resource index numbers range from 0 to 31, and the Media Resource index numbers also range from 0 to 31. The Fax Resource index numbers range from 0 to 3.

To find out how to find the number of Resources available on your system, see either the ADL or ADX online help.

2.1.1.1 Trunk Resources

Trunk Resources are responsible for all call control. Call control includes dialing out, accepting an incoming call, and hanging up when a call is finished. Because a Trunk Resource processes a single stream of audio, each of the following is considered a single Trunk Resource:

- The trunk interface for a single analog line
- One T-1 or E-1 time-slot
- An MSI station
- Simulated Phone's simulated trunk line (always Index number 0)

For example, if you invoke the ADL function TrunkAnswerCall, the Resource Manager layer determines which trunk interface is being used, whether it is legal to make this call (has an incoming call been signaled?), and then which API function to use: *dx_sethook* or *gc_AnswerCall*.

2.1.1.2 Media Resources

Media Resources control the playing and recording of sound files and tones, as well as getting DTMF (Dual Tone Multi-Frequency) digits from callers. For example, a voice recorder that records the caller's response would be considered a Media Resource.

Typically there is one Media Resource that corresponds to each Trunk Resource, but this is not always the case.

2.1.1.3 Fax Resources

Fax Resources control the transmission and processing of fax data. A single fax channel on an Intel Dialogic VFX board or on a GammaLink CP board is considered to be one Fax Resource.

It's important to remember that Fax Resources can only send and receive fax data. All other functions that are required to answer and make telephone calls are performed by Trunk and Media Resources.

2.1.1.4 Voice Recognition Resources

Voice Recognition (VR) Resources translate a caller's spoken input into text strings. One VR Resource can perform a recognition on a single stream of audio data from one Trunk or Conference Resource.

> **Note:** For each speech engine, the Resource Manager creates one Voice Recognition Resource per compatible Media Resource. Note that some speech engines require Media Resources that support CSP, and, depending on your speech engine's limits, you may not be able to use all of these VR Resources simultaneously. If you have more than one

speech engine, you may have more VR Resources than Media Resources.
See the topic titled, Voice Recognition Resource States in either the ADL or ADX online help for more information.

The Resource Manager VR Technologies also include those of several third-parties, for more information about third-party VR Technologies, see either the ADL or ADX online help.

2.1.1.5 Text-To-Speech Resources

Text-To-Speech (TTS) Resources translate text strings into spoken output. One TTS Resource can speak with a single stream of audio data to one Trunk or Conference Resource.

For example, you might create an application in which callers can call in remotely to access their e-mail and request that messages are read to them over the phone.

> **Note:** The Resource Manager creates one Text-To-Speech Resource for each Media Resource on your system. Depending on your speech engine's limits, you may not be able to use all of these TTS Resources simultaneously. See the topic titled "Text-To-Speech Resource States" in either the ADL or ADX online help for more information.

The Resource Manager TTS Technologies also include those of several third-parties, for more information about third-party TTS Technologies, see either the ADL or ADX online help.

2.1.1.6 Conferencing Resources

Conference Resources control individual conferences. Each conference, regardless of how many parties it contains, is controlled by a single Conference Resource.

> **Note:** Sometimes the documentation included with your telephony boards uses the term *conference resource* to refer to the number of parties an MSI or DCB board can place into conferences. In the Resource Manager, a *Conference Resource* controls the logical part of a conferencing board that manages a single conference.

2.1.2 Resource States

For every kind of Resource (Trunk, Media, Fax, TTS, VR, or Conference), the Resource Manager pre-defines a number of states. For example, a Trunk Resource can be in a Ringing or Disconnecting state, and a Media Resource can be in a Playing or Recording state.

The state of a Resource can change in two ways: unsolicited (as a result of an external event), or as a result of a Resource Manager function/method call. An example of an unsolicited state change is a Trunk Resource that is currently Idle transitioning to a Ringing state when an incoming call is signaled. An example of a state change due to a function call is a Media Resource that is currently Idle transitioning to a Playing state in response to a Play Command.

Functions that can change the state of a Resource managed by the Resource Manager are known as Commands. For example, the ADL function MediaPlayFile invokes the Resource Manager Play Command.

Each Resource can have a variety of states such as Busy, Idle, Recording, and so on. The Resource Manager defines the following state parameters for each Resource type:

- The set of possible states
- The set of Commands that can change the state
- The state(s) in which a given Command may be issued
- The state to which a given Command can take a Resource (always one pre-defined state)
- The beginning and ending states of unsolicited state changes

Collectively this information is called a state diagram. Figure 2–3 represents a simplified sub-set of the Resource Manager Media Resource state diagram.

Figure 2–3 Resource Manager Media Resource States

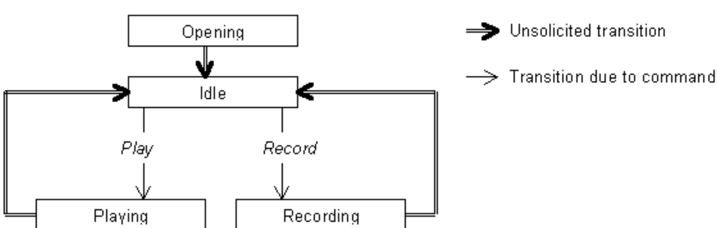


Figure 2–3 shows four states: Opening, Idle, Playing, and Recording, and two Commands: Play and Record. The figure illustrates that it is illegal to issue the Play or Record Command unless the Resource is in the Idle state. If the Play Command is successful, the Resource is always in the Playing state. At some point in the future following a Play Command, the Playing state always transitions back to the Idle state (when the end of the file is reached).

Every Resource Manager Resource must conform to the rules specified in the state diagram for its type. However, a given Resource may implement only a sub-set of Figure 2–3. For example, a loop-start analog trunk (a Plain Old Telephone Service line like you probably have at home) has no signal corresponding to the “out of service”, “out of sync” or “D channel down” conditions found on digital trunks. The Resource Manager defines an unsolicited transition from Idle to OutOfService for Trunk Resources. Digital Trunk Resources may experience this transition, but an analog Trunk Resource will never do it. Just because some Resource Manager Resources support certain state transitions and Commands does not mean that all Resource Manager Resources of that type support them.

A graphical representation of a simple state diagram like Figure 2–3 is easy to grasp. However, for more complex state diagrams this representation can become unwieldy and confusing. Usually, Resource Manager state diagrams are presented in table form. For example, the information in Figure 2–3 can be equally well represented by Table 2–1.

Table 2–1 Resource Manager Media Resource States

Start State	Command	End State
Opening	—>	Idle
Idle	Play	Playing
Playing	—>	Idle
Idle	Record	Recording
Recording	—>	Idle

Unsolicited state transitions are indicated by → in place of a Command.

For more information about state diagrams and tables for each of the Resource types, refer to either the ADL or ADX online help.

2.2 The Resource Manager Profile

The Resource Manager Profile is a database that is organized into a tree of paths and keys similar to the Windows registry structure, and stores this information:

- Details of all installed hardware and API combinations (Technologies)
- User-supplied hardware configuration information
- User-configurable options, such as the default language for speaking values (examples are English and Spanish)

The Resource Manager Profile files are located in the following default installation directory:

```
C:\Program Files\Envox\CT ADE\Common\Config\Profile
```

Unfortunately, the database is not directly readable. In order to view and modify the information in the Resource Manager Profile after running the Resource Manager Scanner utility, use the Profile Configuration Utility. Start the utility in one of the following ways:

- From the **start** menu, select **Programs**, **Envox CT ADE**, **Common**, **Configure CT ADE Profile**.
- Type ProfileConfig.exe -L from a command prompt.

This populates theProfileConfig.txt file which mirrors the database information and is organized in a format that you can read and understand. Each entry in the generated file represents a Profile ID.

You can also view the ProfileConfig.log file to find out what resources were found on your system during a scan as well as any additional files included in the scan.

Both the ProfileConfig.log and the ProfileConfig.txt files are located in the following default installation directory:

```
C:\Program Files\Envox\CT ADE\Common\Config\Bin
```

No dynamically changing information, such as the current state of a Resource, is stored in the Resource Manager Profile. When an application is running it treats the Profile as read-only so the Profile must be fully initialized before an application is started.

2.2.1 Resource Manager Profile IDs

Resource Manager Profile IDs have a name and a corresponding value that describe system details. Internally, the Resource Manager Profile does not store the character strings shown for value names. All value names are actually stored as integers. The set of integers that may be used for value names is pre-defined for a given Resource Manager release. Several Resource Manager utilities convert between the integer values used internally and the string names to display the Profile in a form that is convenient for a human to read. This allows for more efficient lookup algorithms and hence faster access to the Resource Manager Profile when Resource Manager applications are running. The allowed integer values for value names are called Resource Manager Profile IDs.

You can obtain the current value of a Profile ID with the appropriate Get function. For example, in ADL, to find out if ANI (Caller ID) support is enabled, you would use the Trunk GetBool function with REGID_ANISupported (301):

```
Support = TrunkGetBool(301);
```

or

```
Support = TrunkGetBool(REGID_ANISupported);
```

The Profile ID Tables topic lists the Profile IDs that are valid for each Technology.

The name of an entry is similar to a path name in a file system. All names begin at the root, which is designated by a back-slash character (\). For example, a Profile ID much used by Resource Manager code internally is:

```
\TechCount
```

The value of TechCount is the number of different Resource Manager Technologies installed in this PC. Values are one of three types: integer, string, or Boolean (True / False). A convenient shorthand which you may often see shows the value name and value like this:

```
\TechCount=4
```

2.2.2 Resource Manager RegIDs

RegIDs are values used internally by the Resource Manager as well that access technology-level information:

- Some RegIDs are used with the Get functions to retrieve information about the system, and other RegIDs can be used to Set functions to issue a Technology-specific command at runtime. A number of RegIDs can be used for both Get and Set functions. See the CT ADE online help for details of how each specific RegID is used.
- Some RegIDs call underlying API functions. When a RegID causes an API function to be executed, the Resource Manager does not interpret this as a change of state. However, it may change the way that the Resource behaves.
- Some RegIDs describe system details and these are stored in the Resource Manager Profile—these RegIDs are called Profile IDs. You can get the current value of a Profile ID with the appropriate Get function/method. See Section 2.2.1 for details of how to find the current value of a Profile ID. To find out which Profile IDs are valid for each Technology, see either the ADL or ADX online help.

2.2.3 TZP Files

TZP files are also called *Profile Include files* because they hold additional information that the Resource Manager Profile uses to perform various telephony and language functions. CT ADE comes with four TZP files that you can view and modify using a text editor like Notepad.

You can also create your own TZP files, but you must:

1. Save your TZP files with a .tzp extension in the TZP directory. TZP files are located in the following default installation directory:

   ```
   C:\Program Files\Envox\CT ADE\Common\Config\TZP
   ```

2. Add entries to the CTADE.ini file. This ensures that the TZP file gets included during scan time and the information contained in the TZP file is added to the Resource Manager Profile so that the Resource Manager can use it. For more information about using the CTADE.ini file, see Section 2.3.

2.2.3.1 IPFs.tzp File

The IPFs.tzp file stores a set of parameters for each of the fourteen languages that come with the CT ADE installation. Additional IPF files can be added in order to define languages, or simply to take advantage of the performance gains inherent in IPF files.

Figure 2–4 IPFs.tzp File Displayed in Notepad

```
IPFs.tzp - Notepad
File Edit Format View Help
\IPFCount=14

\IPFs[0]\IPFName=English
\IPFs[0]\IPFFile=C:\Program Files\Envox\CT ADE\Common\Config\Language\English Nor
\IPFs[0]\AudioFormatCoding=ADPCM
\IPFs[0]\AudioFormatSamplesPerSecond=6000
\IPFs[0]\AudioFormatBitsPerSample=4
\IPFs[0]\AudioFormatChannelCount=1

\IPFs[1]\IPFName=English_UK
\IPFs[1]\IPFFile=C:\Program Files\Envox\CT ADE\Common\Config\Language\English UK\
\IPFs[1]\AudioFormatCoding=ADPCM
\IPFs[1]\AudioFormatSamplesPerSecond=6000
\IPFs[1]\AudioFormatBitsPerSample=4
\IPFs[1]\AudioFormatChannelCount=1

\IPFs[2]\IPFName=Spanish
\IPFs[2]\IPFFile=C:\Program Files\Envox\CT ADE\Common\Config\Language\Spanish Sou
\IPFs[2]\AudioFormatCoding=ADPCM
\IPFs[2]\AudioFormatSamplesPerSecond=6000
\IPFs[2]\AudioFormatBitsPerSample=4
\IPFs[2]\AudioFormatChannelCount=1

\IPFs[3]\IPFName=Spanish_CA
\IPFs[3]\IPFFile=C:\Program Files\Envox\CT ADE\Common\Config\Language\Spanish Cas
\IPFs[3]\AudioFormatCoding=ADPCM
\IPFs[3]\AudioFormatSamplesPerSecond=6000
\IPFs[3]\AudioFormatBitsPerSample=4
```

The following Indexed Prompt File parameters are also called Profile IDs. For more information about Profile IDs, see Section 2.2.2.

- IPFName—Used by functions or vocabulary entries to specify which IPF file contains a desired prompt.
- IPFFile—Specifies the name and path to the IPF File.
- AudioFormatCoding—Specifies whether the file was recorded with A-law or mu-law companding. Valid values for AudioFormatCoding are Alaw or muLaw.
- AudioFormatSamplesPerSecond—Specifies the sampling rate, measured in samples per second (Hz). VOX files use 6 kHz or 8 kHz, so valid values for AudioFormatSamplesPerSecond would be 6000 or 8000.
- AudioFormatBitsPerSecond—Specifies the amplitude (loudness) of the sound at the instant a measurement was taken represented by a binary number. Sample sizes are generally 4-, 8- or 16-bit. Valid values for AudioFormatBitsPerSample are 4, 8, or 16.
- AudioFormatChannelCount—Specifies whether a file is recorded in stereo, which has a channel count of 2, or mono, which has a channel count of 1

For more information about this file, see Section 2.4.1.1.

Tip: To find out whether your modified TZP file/s got included in the scan, run the ProfileConfig.exe then open the ProfileConfig.log file and check the include files listed under the heading:

```
These files are included in CTADE.ini [ProfileIncludeFiles]
```

2.2.4 Running the Resource Manager Scanner

When you run the Resource Manager Scanner, it creates a database called the Resource Manager Profile.

Also called the Resource Scanner, this utility scans your system hardware and collects information on any installed telephony boards as well as their configuration details. The Scanner then builds a database called the Resource Manager Profile and places the collected information into it.

If you want, you can merge additional information that isn't collected during a scan into the Resource Manager Profile or delete information from it by using *TZP files*.

The Resource Manager Scanner automatically runs upon installation, but you must rerun the Scanner each time you make a change to your current configuration.

To run the Scanner:

1. Run the Resource Manager Configuration Tool from:

   Start | Programs | Envox CT ADE | Common | Configure CT ADE Profile

2. Start the Scanner using either:

   File | Refresh Profile... or
   File | Recreate Profile

   The Resource Manager extracts all the available configuration information from your telephony boards and categorizes these into one of the six Resource API types. The Refresh option will keep changes, while the Recreate option will discard the existing profile and create it new.

3. Next, you need to update the profile with additional information that the Resource Manager cannot automatically determine. Use the Configuration Utility to update the Profile.

   For example, for an E-1/R2 trunk, you must specify which country-dependent parameters should be used for the R2 protocol. For an analog trunk, you must specify whether caller ID is available. For a T-1 trunk, you must specify if ANI and/or DNIS is available, whether ANI/DNIS (Automatic Number Identification /Dialed Number Identification Service) is provided through DTMF or MF digits and how many digits are sent, and so on.

   You can also include TPZ files manually using File | Import.

4. If you've updated one or more TZP files with additional modifications, you need to re-import them to merge the modified TZP file/s into the Resource Manager Profile. If you haven't modified a TZP file, skip this step.

2.3 CTADE.ini File

The CTADE.ini file is used to specify various configuration parameters that are read by the Resource Manager each time it is invoked on your system. If you change any entries in the CTADE.ini file, you must be sure that all instances of ADL/ADX applications and any CT ADE utilities such as the Profile Configuration Utility or Phraser.exe (which use the Resource Manager) are closed before the changes can take effect.

You can edit the CTADE.ini file with a text editor such as Notepad, but the file must reside in the same directory where Windows is installed. Typically this is in the C:\WINNT directory.

Figure 2–5 CTADE.ini File

```
CTADE.ini - Notepad
File  Edit  Format  View  Help
[Profile]
Path=C:\Program Files\Envox\CT ADE\Common\Config\Profile

[ProfileIncludeFiles]
C:\Program Files\Envox\CT ADE\Common\Config\TZP\Languages.tzp
C:\Program Files\Envox\CT ADE\Common\Config\TZP\IPFS.tzp
C:\Program Files\Envox\CT ADE\Common\Config\TZP\Tones.tzp
C:\Program Files\Envox\CT ADE\Common\Config\TZP\Tts.tzp

[ProfileExcludeTechnologies]
TapiMedia
TapiTrunk

[WaveScanner]
WaveDeviceID0=0,0

[SpeechPearl]
PhilipsDir=C:\PHILIPS
```

The CTADE.ini file is populated with several sections by default (see Figure 2–5) but you can also add additional sections for the Resource Manager to use.

CTADE.ini file sections:

- [NFAS]—Lists NFAS boards that don't use a D channel
- [Profile]—Resource Manager Profile startup information
- [ProfileDependencies]—Windows services that must be started before the Profile Configuration utility starts
- [ProfileExcludeTechnologies]—Telephony technologies to exclude from the Resource Manager Profile scan

- [ProfileIncludeFiles]—Include files to be merged with the Resource Manager Profile during AutoMerge
- [SimMediaScanner]—IDs of the Wave In and Wave Out devices you want SimMedia to use
- [SpeechPearl]—Scansoft (Philips) SpeechPearl configuration information
- [SpeechWorks]—Defines DialogModules, services, and vocabularies for the OSRVR Technology
- [WaveScanner]—IDs of the Wave In and Wave Out devices you want WaveMedia to use

2.4 Languages in the Resource Manager

The CT ADE installation comes with fourteen different languages that the Resource Manager uses to translate audible information (Text-To-Speech capability) to a caller. A language consists of a set of recorded words and phrases in a particular language (and possibly gender) along with the supporting files needed for the Resource Manager to find and properly use the prompts. Typical phrases generated by call processing applications might be, *"You have twenty-three new messages,"* or, "*Your current balance is nineteen dollars and one cent."* Phrases consist of pre-recorded sentences or parts of sentences like "You have" and variable information like "twenty-three."

Variable information is constructed by stringing together pre-recorded prompts. For example, the money amount $19.01 would be generated by Resource Manager from the following five prompts:

"nineteen" "dollars" "and" "one" "cent"

By default, your application is able to generate variable information for numbers, money amounts, dates, time, and ordinals in a number of languages using prompts from one or more files that are provided with the default installation.

You can also create your own language if you want. For instructions on how to do this, refer to the ADL or ADX online help.

2.4.1 Language Files

All the information about a language is divided into four main groups:

- IPF (Indexed Prompt File)
- Language Profile
- TZP files
- Optional text files

2.4.1.1 IPF

The Indexed Prompt File (IPF) contains VOX files, also known as prompts, which are recorded words or phrases. Compiled with an .ipf extension, each IPF begins with a header that includes an indexed listing all of the included prompts. For more information about this file, see Section 2.2.3.1.

2.4.1.2 Language Profile

The Language Profile file contains attributes for a language, called vocabularies, as well as rules that govern the playing of those prompts. This set of rules is called a grammar. Stored as an .LNG file, the Resource Manager uses this file to determine which prompts to play and when.

Both the IPF and the Language Profile file are stored in the language default installation directory:

```
C:\Program Files\Envox\CT ADE\Common\Config\Language\<Language_1>
```

2.4.1.3 TZP Files

The Resource Manager Profile Include files, also called TZP files, are a pair of text files containing entries that are added to the Resource Manager Profile via the CTADE.ini file. (The CTADE.ini file contains paths to all the available languages and gets merged into the Resource Manager database at startup time.)

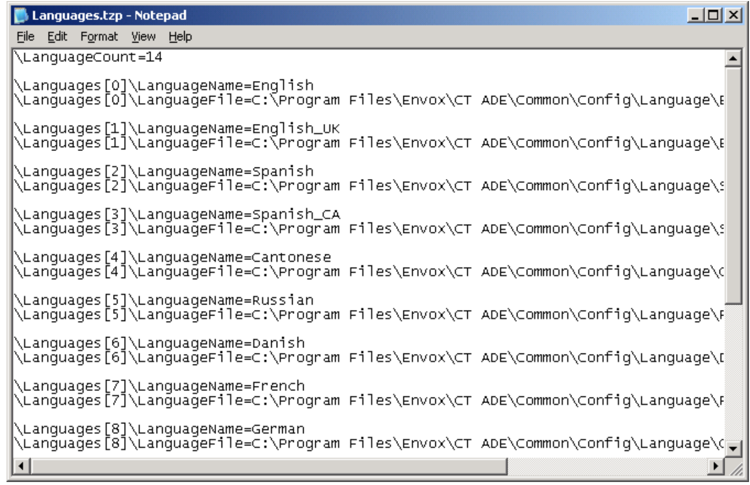
There are two types of TZP files: an IPF include file, and a Language include file. Together they provide the Resource Manager with a "map" to the locations of the IPF and Language Profile for a particular language:

- IPF include file – A TZP file that maps to the location of your language's Indexed Prompt File.
- Language include file – A TZP file that maps to the location of your language's Language Profile

2.4.1.4 Languages.tzp File

The Languages.tzp file designates the same number for each language that the IPF.tzp file does and stores each language's directory location so the Resource Manager can find it on your system. In the following graphic, English is designated the number 0, and its file location is displayed on the next line.

Figure 2–6 Languages.tzp File Displayed in Notepad



The TZP files are located in the following default installation directory:

```
C:\Program Files\Envox\CT ADE\Common\Config\TZP
```

2.4.1.5 Text Files

Optionally, a language can also have two text files that are used to test words and phrases:

- Language.TST file – Text file the Phraser.exe uses to test prompts
- Language.TXT file - Text file listing the names of all your prompts

Both of these files are stored in the language directory along with the IPF and Language Profile file.

2.4.2 Phrase Element Parameters

In the Resource Manager, recorded words are known as elements, which can be strung together to form phrases. For example the phrase *"three hundred twenty-three"* can be broken up into the elements, "three," "hundred," and "twenty-three". There are four parameters required for every phrase element: Value, Type, Gender, and Language. You set these parameters in your application and then Resource Manager uses the parameters to play phrases.

2.4.2.1 Value Parameter

The Value parameter is passed to the Resource Manager and used in conjunction with the Type parameter to provide the element's content. For example, a value of 330 may produce a spoken phrase of *"three hundred and thirty"* or *"three-thirty, p.m."* depending on the Type

parameter used. The Resource Manager determines the Value parameter from the parameters in the ADL Play function/ADX Play method you use.

2.4.2.2 Type Parameter

The Type parameter tells the Resource Manager how to interpret the Value. Some of the built-in Types include Numbers, Ordinals, Dates, Times, and Money. To see how the Type parameter can affect the way element is spoken, let's look at a couple of examples that have the same Value, but different Types.

Value	Type	Spoken
56	Numbers	fifty-six
56	Ordinals	fifty-sixth
123456	Numbers	one hundred twenty-three thousand, four hundred fifty-six
123456	Times	twelve thirty-four and fifty-six seconds p.m.
20020301	Numbers	twenty million, twenty thousand, three hundred one
20020301	Dates	March first, two thousand two

The Play method/function you use determines the Resource Manager phrase Type. In ADL for example, if you call the PlayOrdinal function, the Type is set to Ordinals. If you call the PlayValue function, you can manually set the Type with the ValueType parameter.

In addition to the Types that come with the Resource Manager, you can create your own to accompany an existing language or as part of a new language. For more information about Type parameters, refer to either the ADL or ADX online help.

2.4.2.3 Gender Parameter

Gender refers to the gender to use for the element, that is, the gender of the word itself, not the gender of the voice speaking the prompt. For example, the French for “one book” is “un livre,” which is masculine, so if you were constructing a phrase that gave a number of books, you would set the Gender parameter to masculine. The Resource Manager uses the value of the Gender to determine the second digit in the Grammar label to jump to.

Gender	Value
neuter	0
masculine	1
feminine	2

In ADX, for example, the Resource Manager determines the Gender by reading the ADX Voice Object Gender property, so to speak the word “seven” before a feminine noun, you would use:

```
ADX VoiceObject1.Gender = 2
Call VoiceObject1.PlayInteger(7)
```

In English, the gender will always be neuter since English adjectives aren't gendered.

Gender is an integer value which may be used to modify how a value is spoken, or may be ignored, depending on the grammar file. For example, in English grammars, it is usually ignored since English has no concept of gender. In French, the Gender property would determine whether PlayInteger(1) speaks “un” (Gender=1, masculine) or “une” (Gender=2, feminine). German has three genders, and so on.

Even in English, however, the Gender can provide a “hook” for special purposes with user-defined data types. For example, should a time be spoken using 12- or 24-hour clock?

2.4.2.4 Language Parameter

The Language parameter tells the Resource Manager which Language Profile and IPF to use when processing a phrase element. The Language parameter must match the name of a language name set in the language include file. The Language is a string that gets matched against a language registered in the Resource Manager Profile. If the Language is an empty string, the default will be the first language specified in the Resource Manager Profile.

3

Using Simulated Phone

This chapter describes how to use the CT ADE Simulated Phone with ADL and ADX and contains the following sections:

- Overview
- Starting Simulated Phone
- Incoming Calls
- Phone Line Properties
- Outbound Calls

3.1 Overview

Simulated Phone is a phone simulator application that performs telephone functions for your CT applications and does not require telephony hardware or Intel NetStructure® Host Media Processing Software (HMP software) to be present on your PC. This can be helpful, for example, if you are on a business trip and the only computer available is a laptop without the telephony hardware or HMP software installed.

Simulated Phone works with both ADX and ADL applications, including ADL applications running in the Debugger. You can also demonstrate a finished application to a client using Simulated Phone and a standard PC with a sound card.

Simulated Phone icon

Simulated Phone supports a variety of Vox and Wave formats. It can also play and record sound files in real-time and with high quality using your SoundBlaster™ or other Wave-compatible sound card. For more information about playing and recording files, see either the ADL or ADX online help.

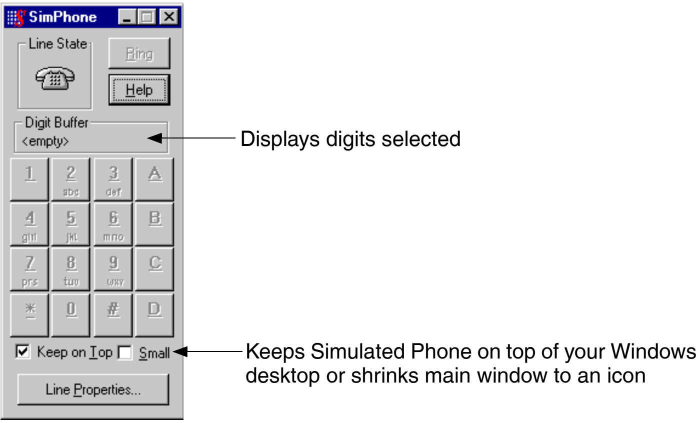
3.2 Starting Simulated Phone

To launch the Simulated Phone application, select Simulated Phone from the Start menu. In ADL Studio, you can also start Simulated Phone from the main menu by choosing Run | Launch Simulated Phone. Simulated Phone always simulates Trunk Resource 0 and Media Resource 0.

For more information on Trunk Resources or Media Resources, refer to the ADL or the ADX online help.

> **Note:** If you start Simulated Phone when an application is already running, your ADX application will continue to use the telephony card.

Figure 3–1 Simulated Phone at Startup



After you've started Simulated Phone, it's time to start your application.

3.3 Incoming Calls

Most applications wait for an incoming telephone call. You can send a call to your application by clicking the Ring button. This triggers an incoming call event to Simulated Phone just like a ring on a telephone line would. Your application usually responds by answering the call. Simulated Phone displays the current state of the simulated telephony channel's hook switch by the Line State icon: when the line is off hook, the off-hook icon is shown, and when the line is on hook, the on-hook icon is shown.

Line State Icon	Description
	Simulated phone line is on hook. If the icon is grey, Simulated Phone is not being used
	Simulated phone line is off hook

When the line is off hook, the Line State icon shows a telephone with the handset picked up (off hook). The *close* system button and menu option are disabled (they appear grey instead

of black), the Ring button becomes the Hangup button, and each DTMF (Dual Tone Multi-Frequency) Digit button is enabled. Clicking the Hangup button simulates a caller hang-up.

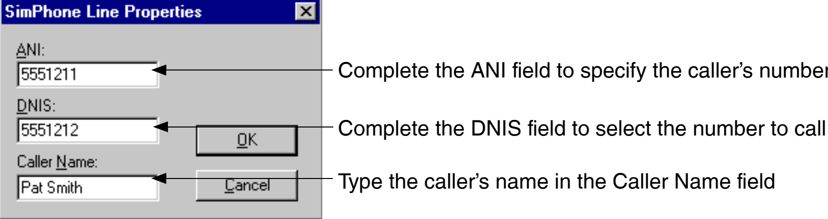
Pressing digit buttons has the effect of adding the digits to the digit buffer on the simulated channel and the current contents of the buffer are shown in the Digit Buffer display. The buffer will be reduced or emptied when the appropriate ADL functions/ADX methods are invoked.

You cannot exit Simulated Phone while an application is in progress. This is indicated by the greying of the close system button and menu option. This button will be enabled again when you stop your application.

3.4 Phone Line Properties

Simulated Phone lets you set the ANI (Automatic Number Identification), DNIS (Dialed Number Identification Service), and caller name that are reported to your application when you make an inbound call. Before placing a call to your application, you must set the phone line properties by clicking the Line Properties button at the bottom of the Simulated Phone main window, and then completing the appropriate fields.

Figure 3–2 Simulated Phone Line Properties Dialog



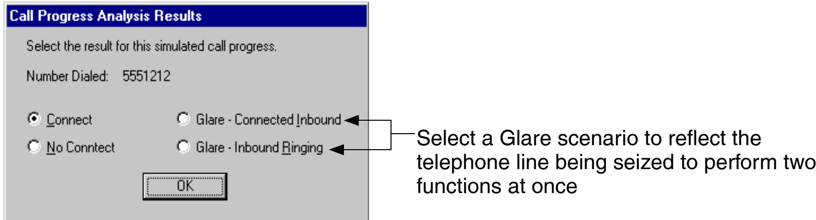
Click OK, and the new settings take effect with the next call to your application.

3.5 Outbound Calls

Simulated Phone can also receive outbound calls from ADL or ADX applications.

When your application places an outbound call that requires Call Progress Analysis (for example, if you use the Dial with Analysis cell in a flow chart or if you use the TrunkMakeCall function in the ADL language), Simulated Phone displays a dialog that allows you to specify the result of the call. This is shown in Figure 3–3.

Figure 3–3 Call Progress Analysis Results Dialog



Select one radio button to indicate what call scenario you want Call Progress Analysis to return. Click OK, and Simulated Phone simulates the result you chose.

For more information about Simulated Phone, refer to the Simulated Phone online help.

4

Getting Started with ADX

This chapter introduces ADX and contains the following sections:

- ADX Components
- ADX Programming

Refer to the samples in *...\Program Files\Envox\CT ADE\ADX\Samples* for more information.

4.1 Overview

The ADX portion of Envox CT ADE™ provides a specialized platform that lets you reduce the repetitive tasks associated with a traditional telephony application development, and enables you to stay focused on the essential and profitable aspects of developing your application.

Made up of a collection of ActiveX components, ADX makes it easy for you to add CT functions to your existing systems or to create applications from scratch. The ActiveX components can be used in any Windows development environment that supports ActiveX components.

You develop applications in ADX by writing your application directly in the programming language of your choice and adding ActiveX components to it.

4.1.1 ADX Components

The ADX set of ActiveX components can integrate directly into visual programming environments such as Visual Basic .NET, Visual C#, or Delphi, and provide CT-specific development functionality.

ADX components are invisible at runtime and do not have an interface, but you can easily create one using the conventional visual controls such as buttons, scroll bars, etc., so that your system administrator can more easily facilitate the application during runtime.

4.1.1.1 ADX Voice Object

ADX Voice Object gives you control of trunk and media hardware through Resource Manager Trunk Resources and Media Resources by handling call control, the playing and recording of sound files and tones, and getting DTMF digits from callers.

Trunk Resources are responsible for all call control, which includes dialing out, accepting an incoming call, and hanging up when a call is finished. Media Resources control all playing and recording of sound files and tones, as well as getting DTMF digits from callers.

4.1.1.2 ADX Fax Object

ADX Fax Object adds fax support to your applications by giving you control of fax hardware through Resource Manager Fax Resources. ADX Fax Object works closely with ADX Voice Object and the other ADX controls. In your programs, you use ADX Voice Object first to control call processing functions such as answering or placing phone calls. Next, you use ADX Fax Object to process the fax information. Finally, you use ADX Voice Object again to finish processing the call.

4.1.1.3 ADX Conference Object

ADX Conference Object lets you manage conferencing capabilities in your applications. You also control conferencing hardware through Resource Manager Conference Resources. ADX Conference Object works closely with ADX Voice Object and the other ADX controls. In your programs, you use ADX Voice Object first to control call processing, such as answering or placing phone calls, playing menus to callers, and getting the digits they type in response. Next, you use ADX Conference Object to connect to a conference, and then to disconnect from the conference. Finally, you use ADX Voice Object again to finish processing the call.

4.1.1.4 ADX TTS Object

ADX TTS Object lets you control text-to-speech features in your applications through Resource Manager Text-To-Speech Resources. ADX TTS Object works closely with ADX Voice Object and the other ADX components. In your programs, you use ADX Voice Object first to control call processing, such as answering or placing phone calls, playing menus to callers, and getting the digits they type in response. Next, you use ADX TTS Object to speak text to the caller. Finally, you use ADX Voice Object again to finish processing the call.

4.1.1.5 ADX ASR Object

ADX ASR Object lets you add speech recognition capabilities to your application through Resource Manager Voice Recognition Resources. ADX ASR Object works closely with the ADX Voice Object control. In your programs, you use ADX Voice Object first to control call processing, such as answering or placing phone calls, playing menus to callers, and getting the digits they type in response. Next, you use ADX ASR Object to recognize speech from the caller. Finally, you use ADX Voice Object again to finish processing the call.

4.1.1.6 Media Toolbox

Media Toolbox is a family of ActiveX components that you can use to further manipulate telephony sound files. After installing the utility you have access to a host of tools including a sound file converter component, a component that helps you create and edit Indexed Prompt Files (IPFs), a component that can query the capabilities of your installed Wave devices, and sample applications.

4.1.2 ADX Programming

As you develop and run applications using ADX, you use methods to send commands to the Resource Manager, define properties to describe the attributes of the various objects you are using, and monitor events to find out what actions have occurred during development and runtime.

4.1.2.1 Methods

Methods are functions that implement most actions in your program. You send commands to the Resource Manager requesting specific CT functions.

You can run ADX in synchronous or asynchronous mode. In synchronous mode, methods put the calling task or thread to sleep (this is called blocking) until the requested operation completes.

In contrast, when asynchronous mode is enabled, supported methods return immediately and indicate whether the operation started successfully while the operation continues in the background. Later, your application receives an event reporting that the operation has completed. Asynchronous mode is disabled by default. Many (but not all) ADX methods are affected by asynchronous mode; to find out which methods are affected, see the ADX online help.

4.1.2.2 Properties

A property is a data value similar to a variable and is generally used for attributes of an object that tend to remain fixed. In ADX, for example, the ElapsedSecs property returns the number of seconds that elapsed during the last Play or Record. Properties are specified using the name of the object and the name of the property separated by a period. You set property values at runtime.

4.1.2.3 Events

Events are reports from the object to its client signifying that an action has occurred. Many events have corresponding methods that wait for a specified action to take place. For example, if an object displays a button on the screen, a typical event may report that the button was clicked. In ADX, you can process events by defining a function to be called when the event is triggered.

5

ADL Concepts

This chapter gives you an overview of the most important concepts of ADL and contains the following sections:

- Overview
- Writing Simple Expressions
- Forming Complex Expressions
- Using Statements
- Constructing Loops
- Utilizing Functions

5.1 Overview

ADL is easy to learn, and you can create robust applications quickly. Specially designed for call processing, ADL uses very little dynamic memory allocation so live systems are very unlikely to run out of memory.

More simple to use than Visual Basic and C/++, ADL borrows some of the best ideas from the C language, but avoids many of the problems that make using C difficult for beginners (such as syntax) and experts (such as data types and dynamic memory allocation).

If you are a beginner, you'll find ADL easy to learn; and if you're an expert who knows C, you'll quickly adapt to the changes relative to C (which are designed to help beginners avoid common mistakes).

5.1.1 Source Code Basics

As you write ADL programs, keep the following rules in mind:

Differentiate Between Upper- and Lower-case Letters

ADL is a case-sensitive language. All parts of the language recognize the difference between upper-case letters (ABC...) and lower-case letters (abc...).

Begin Comments with a Pound Sign (#)

You can add comments to your code by typing a pound sign (#) at the beginning of a line. Comments continue up to the end of that same line.

The pound sign may appear as £ or another special character on non-US PCs; it is the ASCII character with code 35 decimal, 23 hex.

Use End-of-line Marks (;)

The end-of-line marks have no syntactical significance except that they terminate a line of code.

Multiple statements may be included in a single line, although this is discouraged because it generally makes the source code harder to read.

You can find more detailed information about ADL language in the ADL online help.

5.1.2 Source Code Example

The basic structure of ADL flows like this:

```
# This program is called inbound.vs.
program
  TrunkWaitCall();
  TrunkAnswerCall();
  if (TrunkGetState() strneq "Connected")
       voslog("Unexpected trunk state ", TrunkGetState());
       stop;
  endif
  InboundCall();
  TrunkDisconnect();
  restart;
endprogram
```

5.2 Writing Simple Expressions

Simple expressions consist of a single value. There are three kinds of simple expressions: values, variables, and constants.

5.2.1 Values

All values in ADL are stored as character strings. Character strings are written as a sequence of characters inside double quotes:

```
"This is a string"
```

5.2.1.1 Numerical Values

Numbers are represented by strings of decimal digits. For example, one hundred and twenty-three is represented as the string of three characters “123.” You can omit the double quotes when writing a number, so both of these examples are acceptable ways of writing numbers:

```
"123"
123
```

5.2.1.2 Logical Values

True and False values are also represented as strings. False is represented as an empty string containing no characters, written like this:

```
""
```

True is represented as a string containing the decimal digit 1:

```
"1"
```

5.2.2 Variables

A variable has a name and contains a character string. All variables are set to empty strings when ADL starts and when a restart statement is executed. Variables must be declared, meaning that you must give them a name and maximum length before ADL can use them. The maximum length cannot exceed the current maximum string length or ADL truncates it. See the topic “MaxStringLength” in the ADL help for more information. You can declare variables in two places: before the start of the main program, or inside of a function.

If you are running ADL in Debug mode, a warning message is issued if a string is truncated by assigning it to a variable. To avoid this warning, assign the string to a variable using a substr function, which is an ADL function that returns a substring of a given string.

Why a maximum length for each variable? Why doesn’t ADL allocate memory dynamically? This is because call processing systems must often run unattended for days, weeks, and months at a time. ADL is designed to avoid dynamic memory allocation in all areas to avoid problems caused by memory fragmentation and running out of memory in a live system. This is one of many reasons that we chose to design a new language for ADL rather than use an existing language.

> **Note:** One of the most common programming mistakes that ADL beginners make is forgetting that variables have a fixed length and that values longer than the fixed length are truncated. This is another good reason to make a habit of using ADL in Debug mode and to look at the ADL.LOG file on a regular basis to make sure that there are no error messages or warning messages.

5.2.2.1 Example Program with a Variable Declaration Block

In this example, two variables are declared: one named x, with a maximum length of two characters, and one named y, with a maximum length of three characters.

```
dec
  var x : 2;
  var y : 3;
enddec

program
  x = 12;
  y = 123;
  y = "Too long"; # y becomes "Too"
endprogram
```

A variable name must start with a letter and may continue with any number of letters, digits (0 through 9) and underscore characters (_). Because the ADL language is case sensitive, the variable names X and x refer to two different variables.

5.2.3 Constants

A constant is a named value. That is, a name assigned to a fixed string of characters. Constants are declared inside a dec...enddec block like this:

```
dec
  var x : 2;
  var y : 3;
  const LINE_NUMBER = 1;
  const MAX_TIME = 60;
enddec
```

The name of a constant may be used anywhere a value is expected in the language. For example, you can use a constant to specify the maximum length of a variable:

```
dec
  const MAX_DIGITS = 7;
  var PhoneNumber : MAX_DIGITS;
enddec
```

5.3 Forming Complex Expressions

You can create more complex expressions using operators. Operators take one, two, or three values and produce a single value as a result. There are two kinds of operators, arithmetic operators and logical operators.

5.3.1 Arithmetic Operators

A familiar example of an operator is the plus sign (+), called the addition operator. It combines its left-hand value and its right-hand value to give a single result. ADL includes a set of six arithmetic operators. Similar to algebra, operators have different strengths called *precedences*.

Operation	Expression
Add	`Left + Right`
Subtract	`Left - Right`
Multiply	`Left * Right`
Divide	`Left / Right`
Modulus	`Left % Right`
Change sign	`-Right`

For example, multiplication is stronger than addition, so in the expression A+B*C, the multiplication B*C is performed first and the result added to A. Parentheses () may be used to create groups and force evaluation in the desired order, so in the expression (A+B)*C, the addition will be performed first.

> **Note:** Arithmetic is performed internally by ADL using 32-bit integer precision. Results involving 10 or more decimal digits may be incorrect. No warning or error is given by ADL when a value overflows 32-bit precision.

Because ADL uses 32-bit integers internally, values from -2^{31} to 2^{31}-1 can be represented, which is the range:

```
-2147483648 to 2147483647
```

This means all 9-digit decimal integers can be represented, but most 10-digit values cannot. For example, that you can always add two 9-digit numbers, so this works correctly:

```
999999999 + 888888888
```

However, this doesn't work because the intermediate multiplication result overflows 32 bits:

```
(999999999*5)/4
```

5.3.1.1 Arithmetic Using Decimal Points

ADL has no built-in functions for dealing with fixed-point or floating-point decimal arithmetic. The Fixed Point Math RLL, which is included in the default installation for ADL, provides decimal arithmetic functions fp_add, fp_sub, and so on. However, ADL applications can still work easily with decimals such as dollars and cents without needing to load an RLL. The easiest way to do this is to multiply all values by 100 (this is for precision

to two decimal places; if you need precision to three decimals, multiply by 1000, and so on). You can then add, subtract, and compare values using these new units (corresponding to cents, tenths of cents, etc.) and convert back to dollars and cents as necessary when you are done. In the case of money values, this corresponds to keeping all values in cents.

Converting Money Values Between Cents Only and Dollars and Cents

Let’s suppose that **TotalCents** contains a money value as cents:

```
Dollars = TotalCents/100;
Cents = TotalCents % 100;
```

If you have two variables containing dollars and cents, they can easily be converted to a total cents value:

```
TotalCents = Dollars*100 + Cents;
```

If you have a string, say in a variable named Dec, containing a decimal point character, you can convert to dollars and cents like this:

```
PosOfPoint = strpos(Dec, ".");
if (PosOfPoint eq 0)
 Dollars = Dec;
 else
 Dollars = substr(Dec, 1, PosOfPoint-1);
 Cents = substr(Dec, PosOfPoint+1);
endif
```

5.3.2 Logical Operators

ADL includes three logical operators and eight comparison operators that are used to include, exclude, or compare values to give a logical result (True or False) result. True is represented as “1”, and False is represented as “”. Logical operators that combine two True/False values:

Operation	Expression
Logical AND	Left and Right
Logical OR	Left or Right
Logical NOT	not Right

5.3.3 Comparison Operators

Comparison operators that combine values and give a logical (True/False) result:

Operation	Expression
Greater than	`Left > Right`
Greater than or equal	`Left >= Right`
Less than	`Left < Right`
Less than or equal	`Left <= Right`
Equal numerically	`Left eq Right`
Not equal numerically	`Left <> Right`
Equal as string	`Left streq Right`
Not equal as string	`Left strneq Right`

Note: There is a subtle difference between the equality of numbers and strings. Strings are converted to a number by taking all the characters up to the first non-digit. So this means that although "123", "0123", and "123ABC" are all equal numerically, they are different when compared as strings. This is important to remember when making menus using the pound or star keys because we are comparing zero with zero:

```
      "*" eq "*" # This gives True
"#" eq "*" # This also gives True
```

Be sure to use streq, not eq, when comparing non-numeric values.

Logical conditions can include arithmetic operators and parentheses just like other expressions. In fact, ADL makes no distinction between logical conditions and other expressions except to convert the final result to a logical value rather than a numerical value. Logical conditions are used in if-statements and to control the following kinds of loops: for, while, and do...until.

For example, this is a logical condition used in an if-statement:

```
if (Year eq 1996 and Month > 6)
```

5.3.3.1 Assignments

The assignment operator = is a special kind of operator. The left side of the operator must be a variable or array element (see the ADL online help for more about arrays), and the right side is any expression.

Assignment Operator Examples

The following are examples of valid assignments:

```
x = 1;
Month = 3;
MonthName = "March";
TotalSeconds = Hours*3600 + Mins*60 + Secs;
```

There are other assignment operators which can be useful: +=, -=, *=, %=, and /=. The one most often used is +=, which could be described as *add to*. For example, this adds 1 to n:

```
n += 1;
```

The expression on the right-hand side is evaluated, and the value is added to the variable on the left. In a similar way, -= is *subtract from*, and so on. Since adding and subtracting 1 is so common, further short-hands ++ and -- are provided. They work like this:

x++	Add one to x, result is x before adding one.
++x	Add one to x, result is x after adding one.
x--	Subtract one from x, result is x before subtracting.
--x	Subtract one from x, result is x after subtracting.

You don't have to be concerned about the result if all you want to do is add or subtract one from a variable. For example, the two alternative statements have exactly the same effect:

```
x++;
++x;
```

The result matters if you use the result later in executing a statement:

```
x = 8;
y = x++;# y=8, x=9
x = 8;
y = ++x;# y=9, x=9
```

Notice that different values are assigned to y. If you find +=, ++, and similar expressions confusing, you can always achieve the same result by simply using =. Although if you're a C programmer and find them convenient, go ahead and use them.

5.3.3.2 String Concatenation

The & operator combines two strings by placing the characters on the right side after the characters of the left side. This is called string concatenation. For example, A & B gives AB. The following example results in FullName being assigned Joe Smith:

```
FirstName = "Joe";
LastName = "Smith";
FullName = FirstName & " " & LastName;
```

Note: String concatenation is one of the strongest operators. This can lead to subtle errors when used together with arithmetic operators. See the following example to find out how to avoid this kind of error.

Example of a String Concatenation Operator

This expression assigns 0 to x:

```
x = "Result is " & 2*2;
```

This is because & is stronger than *, so concatenation is done first, giving the result:

```
x = "Result is 2"*2
```

But *Result is 2* is zero when converted to a number (because the first character is not a digit), so the final expression becomes 0*2 = 0. To make sure that the multiplication is done first, you can use parentheses:

```
x = "Result is " & (2*2);
```

5.4 Using Statements

There are seven basic kinds of statements: if statements, switch/case block statements, goto statements, jump statements, include statements, stop statements, and restart statements.

5.4.1 Switch/Case Statements

A *switch/case block* statement is a way to make a decision between several different options. If ... else ... endif is a one- or two-way decision, but switch/case allows for multiple decisions. A typical example where switch/case is useful is with touch-tone menus where the caller may press one of several different choices.

The basic idea behind a switch/case statement is that an expression is matched against several possible values. When a matching value is found, the following statements are executed.

Switch/Case Statement Example

```
Digit = sc_digits(line);
switch (Digit)
case 1:
 One();
case 2:
 Two();
case 3:
 Three();
case "*":
 Star();
```

```
default:
  Invalid();
endswitch
```

One(), Two() ... Invalid() are functions which are defined elsewhere.

The expression in parentheses following switch is evaluated. The value is then compared with the expression following each case until a match is found. The first time a match is found, the statements following that case are executed until the next case, default, or endswitch is encountered. At that point, execution is transferred to the first statement following endswitch. If no matching cases are found, the statements following default are executed. If no matching cases are found and there is no default, then the entire switch...endswitch block is skipped.

Note that the comparisons are made as if streq had been used, not eq.

Full expressions, not just constants or variable names, are permitted following switch and following each case. As in other ADL language constructs, switch..endswitch statements may contain further switches, loops, etc. as required (although this is usually bad style—deep nesting usually results in hard-to-read code so we recommend using functions to break up such code into easy-to-read pieces).

5.4.2 Goto Statements

Goto statements are used to branch to a given location within your program. The target of a goto is a label, which is a name followed by a colon. A label may be applied to any statement.

Goto Statement Example

The following goto statement shows another way to compute a sum (Repeat and Done are labels):

```
n = 1;
Repeat:
  if (n > Max)
      goto Done;
  endif
  Sum = Sum + n;
  n = n + 1;
  goto Repeat;
Done:
  vid_write("Sum = ", Sum);
```

The goto statement is written where *labelname* is defined somewhere in the same program section (main program or given function):

```
goto labelname ;
```

When a goto statement is executed, control is transferred to the statement with that label and execution continues from there. The label may be before or after a goto which references that label. You can't use a goto statement to branch from one function to another or from a function to the main program. You also can't apply a label without having a statement. For example, it is illegal to have a label immediately in front of an endfor keyword:

```
for (;;)
 goto EndOfLoop;
EndOfLoop:# Illegal!
endfor
```

Instead, you can achieve the desired effect by adding an empty statement that consists of just a semi-colon:

```
for (;;)
 goto EndOfLoop;
EndOfLoop:; # With semi-colon is OK
endfor
```

5.4.3 Jump Statements

Like Goto statements, *jump* statements are used to branch to a given location within your program. A jump statement transfers control to a special label in the main program. This may be done from anywhere in the program. If a jump is executed from within a function, the stack is cleared, which means that the memory of all function calls up to that point is erased. A jump label is defined using a label statement that may be applied to any statement in the main program:

```
label labelname :
```

The jump statement looks like this:

```
jump labelname ;
```

Jumps are useful for features like "*press star to return to the main menu,*" which can be very awkward to implement using other flow-of-control statements.

5.4.4 Include Statements

A source code file may include the complete contents of another file by using the *include* statement. This is typically used to include standard lists of constant declarations. For example, the keyword *include* is followed by the name of a file in double-quotes:

```
dec
 include "project.inc"
 var MyVar : 2;
```

```
enddec
```

An include keyword may occur anywhere in the source code, and is replaced by the contents of the referenced file. If the file plus.inc contains the single character +, then:

```
x = y include "plus.inc" z;
```

is equivalent to:

```
x = y + z;
```

You can specify a search path for include files with the [IncludeFileDirs] section of the ADL Settings file.

5.5 Constructing Loops

You can use a loop to repeat one or more statements. The ADL language includes three types of loops: for, do...until, and while. The choice is mostly a matter of style and taste. Any given task that needs a loop can be written using any of the three types.

The italicized text in the following loop examples represents the following:

- *Statements*—One or more statements (which may themselves be loops) that are executed zero or more times as the loop repeats
- *Initialize*—An expression that is executed once before the for loop starts
- *Test*—A logical condition that is evaluated each time through the loop and determines whether the loop continues executing
- *Increment*—An expression that is executed once at the end of each iteration through the for-loop

5.5.1 For Loop

The *for* loop begins by testing for a condition that determines whether to execute the loop operation. If test is True, the loop executes, followed by the increment statement. If the test is False the loop doesn't execute. So if the test is False the first time through, the statements inside the loop are never executed and the increment is never executed.

The test may be left blank, in which case the value is always assumed to be True and the loop repeats forever, or until exited by means of a goto, jump, or return statement. You can leave out the initialize and increment expressions if they are not required.

```
for (initialize ; test ; increment)
  statements
endfor
```

5.5.2 While Loop

Like the for loop, the *while* loop also begins by testing for a condition that determines whether to execute the loop operation. If the test is True, the loop continues to run. If the test is False the loop stops executing so if the test is False the first time through, the statements inside the loop never execute. This loop differs from the *for* loop in that the condition is tested before each loop.

```
while ( test )
  statements
endwhile
```

5.5.3 Do...Until Loop

The *do...until* loop begins by executing the operation and then evaluates the condition that determines whether the operation needs to execute again. This type of loop continues until the test is True. This differs from the *while* loop since the loop will always execute at least once.

```
do
  statements
until ( test );
```

5.5.4 Using loops

The most common example of a loop is stepping through all values of a variable from 1 to an upper limit, say Max. The following loops all compute the sum 1+2+...+Max using a variable called n:

```
Sum = 0;
for (n = 1; n <= Max; n++)
  Sum = Sum + n;
endfor

Sum = 0;
n = 1;
while (n <= Max)
  Sum = Sum + n;
  n++;
endwhile

Sum = 0;
n = 1;
do
```

```
  Sum = Sum + n;
  n++;
until (n > Max);
```

If you don't like the short-hand n++, you can use n = n + 1 or n += 1 instead. Also, Sum = Sum + n could be replaced by Sum += n if you prefer.

A loop that repeats forever (or until exited by a goto, jump, or return command) may be written using a for-loop:

```
  for (;;)
  # ...
endfor
```

—or a while-loop:

```
while (1)
  # ...
endwhile
```

Remember that True is represented as "1", so the *while* test is always True.

5.6 Utilizing Functions

A *function* is a sequence of statements that has a name and produces a value. Functions can have one or more *arguments,* also called *parameters*. Arguments are values that are copied into the function. Inside the function, these arguments behave like variables and are accessed as normal by using their names. However, there is one difference between arguments and variables—an argument may not be assigned a new value.

There are three types of functions in ADL:

- Built-in functions—These functions are hard-coded into ADL and the ADL Compiler, and are always available to be used in a program.
- User-defined functions—These are functions you write in ADL source code.
- RLL functions—These functions are written in C or C++. They are loaded from binary files called Runtime Link Libraries (RLL). In Windows, an RLL is a DLL.

The syntax for calling a function is the same for all three function types. The name of the function is given, followed by a list of arguments in parentheses, and separated by commas. Even if there are no arguments, the parentheses must still be given. Some examples of calls to functions are:

```
MediaPlayFile("Hello.vox");# Built-in function
MyFunc();# User-defined
code = ASUse(recordset);# RLL
```

Because the syntax is exactly the same, you can't tell from the call to the function whether it is built-in, user-defined, or contained in an RLL (although most built-in functions display as highlighted blue in ADL Studio).

When a function name is used in an expression, the function is executed (this is known as calling the function) and the value produced by the function (called the return value) is obtained. The name of the function and its list of arguments are thrown away and replaced by the return value.

All functions return values. If no return value is specified, an empty string (a string containing zero characters is written as "") is returned. Many functions, such as trace, don't return any useful value—trace always returns an empty string. If the return value is not used in the statement, then ADL simply ignores the return value:

```
trace(2); # Return value not used
```

If a useful value is returned, it may be used in an expression. For example, if a function named MyFunc requires two arguments and returns a value, then the following are valid statements:

```
MyFunc(1, 2); # Ignore return value
x = MyFunc(1, 2); # Store return value in x
# Use return value in expression:
y = x*MyFunc(1, 2) + z;
```

The arguments to a function may be constants, variables, or expressions. The following are also all valid:

```
MyFunc(1, 2); # Constants
MyFunc(x, y);# Variables
MyFunc(z, y+123); # An expression
```

An expression used as a function argument may itself contain function calls. Like most other features of ADL, nesting or recursion to many levels is permitted.

5.6.1 Built-In Functions

ADL includes an extensive range of built-in functions for you to use. This list describes some of the most important groups of built-in functions. For a complete list, see the ADL online help.

Trunk—The Trunk functions control Trunk Resources, which let you make outbound calls, answer or reject incoming calls, and get information on incoming calls, such as Caller ID (ANI), DNIS, and Caller Name.

Media—The Media functions control Media Resources, which let you play and record sound files, get digits from the digit buffer, and play tones.

Fax—The Fax functions control Fax Resources, which support fax transmitting and receiving on a call previously established on a trunk.

Conference —The Conference functions control Conference Resources, which support conversations with three or more participants.

Database—The ADO RLL is the recommended way to add mission-critical database functionality to your ADL program. This RLL supports Microsoft ActiveX Data Objects, including extensive support for SQL, transaction processing, multi-user locking and other important features. The built-in db_ functions support database and index access using dBase-compatible DBF database files and Clipper-compatible NTX indexes. These are useful for small ad-hoc databases, especially read-only or rarely-updated configuration information.

Serial Communication—The ser_ functions support RS-232 serial communication through COM ports.

ADL Box—The vid_ and kb_ functions provide for simple screen output and user input to an ADL program. These functions are most useful when developing and debugging because they provide “quick and easy” ways to get data in and out of your program interactively.

User Interface—You may want to create a graphical user interface. You can do this by using the NetHub RLL / ActiveX plus a tool such as Visual Basic, Delphi, Visual C++ or other Windows visual tool to create the GUI itself.

Task Management—There are several built-in functions for managing tasks, including spawn, chain, exec, arg, getpid, and kill.

Inter-Task Communication—ADL provides four methods for communicating between tasks: global variables (glb_ functions), semaphores (sem_ functions), messages (msg_ functions), and groups (grp_ functions).

String Manipulation—There is an extensive set of functions for manipulating strings, including length, substr and many others.

File Access—The fil_ family provides functions for opening, reading, writing, seeking and locking files.

5.6.2 User-Defined Functions

You can define your own functions in ADL. Function definitions appear following the endprogram statement that ends the main program.

Let’s define a simple function named Add2 which adds two numbers (while this isn’t a particularly useful example, it clearly illustrates how to define a function):

```
func Add2(Arg1, Arg2)
 return Arg1 + Arg2;
```

```
endfunc
```

The function definition starts with func, is followed by the name of the function, and then by a list of zero or more parameter names separated by commas. The function definition ends with endfunc.

The return statement exits the function. There may be any number of return statements which may appear anywhere within the function.

A return statement may look like this:

```
return;
```

—in which case an empty string is returned or an expression may be given:

```
return expression;
```

Here, the expression is evaluated and the resulting value is returned. In our example, we want to add the two arguments Arg1 and Arg2, so Arg1+Arg2 is the expression we need.

If no return statement is given, an empty string is returned when execution reaches the endfunc statement.

Any series of statements that would be valid in the main program can be included in a function (except the label statement).

Optionally, a dec...enddec block may be included immediately following the func statement. This is used to declare variables and constants for use only within the current function.

Example of a dec...enddec Block After a Function

```
func Add2(Arg1, Arg2)
 dec
 var Sum : 10;
 enddec

 Sum = Arg1 + Arg2;
 return Sum;
endfunc
```

Function arguments are passed by *value.* This means that a string value is copied into each argument name (Arg1 and Arg2 in our Add2 example) when the function begins execution. You can’t assign a value to an argument:

```
Arg1 = 1; # Illegal!
```

This means that you can’t change the value of a variable by passing the name of the variable as a function argument. This is because it is the value stored in the variable, not the variable name, that is copied into the function. If you need to change variables passed into a function, you can use the *unary indirection* operators & and *. The expression &v is the internal ADL

variable number of v, the expression *v is the variable whose internal number is v. These two can be used to pass variables *by reference* into a function. See the ADL online documentation for more details.

5.6.3 Function Files and Libraries

ADL language functions may appear in separate files. If a function has not been defined in the main source code file, the ADL Compiler searches for an external file containing the function. For example, if a function named MyFunc has not been defined, ADL Compiler searches for a file named MYFUNC.FUN. The first eight characters of the function name are converted to upper case, and the extension .FUN is appended. You can specify a search path for function files with the [FunFileDirs] section of the ADL Settings file.

A collection of Function Files can be combined into a Function Library. Function libraries are created using the mkvl utility. Function libraries can be encrypted so that you can distribute library files without disclosing your source code. See the ADL online help for mkvl and dmpvl for more details.

5.6.4 RLL Functions

Runtime Link Libraries (RLLs) are extensions to ADL written in C or C++. An RLL contains one or more functions that can be called from an ADL application. A function in an RLL can be used exactly like a built-in function in ADL. The function may take either a fixed or variable number of arguments.

RLLs are specified in the ADL Settings file. Each RLL is loaded by ADL or ADL Compiler each time they are run. In a Windows environment, RLLs are implemented as DLLs (Dynamic Link Libraries).

To create your own RLLs you can use C/C++ along with the ADL C Developer's Toolkit and a Microsoft compiler.

> **Note:** If more than one RLL is used, ADL must load RLLs in the same order that ADL Compiler used when compiling the application. This requires that RLLs are specified in the same order in the Settings file used by ADL Compiler and ADL. Of course, using the same Settings file for both ADL and ADL Compiler is the easiest way to achieve this.

RLL functions can be logged like built-in functions by using the trace(8) function or by setting the Override and RLLs entries in the [Trace] section of the ADL Settings file.

6

Writing ADL Script

This chapter begins with a tutorial that teaches you how to create two programs using ADL. Next, it analyzes the code from one of the programs you create. Finally, it provides some commonly used telephony functions written in ADL. It contains the following sections:

- Starting the ADL Program Tutorial
- Examining the Echo Program
- Using Basic Telephony Functions

6.1 Starting the ADL Program Tutorial

Before beginning this tutorial, make sure ADL is installed on your machine correctly. You are also going to use Simulated Phone, which is automatically included in the CT ADE installation.

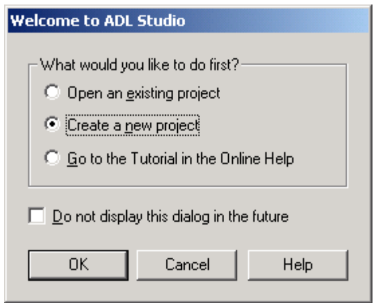
If you have questions or want more information on any of the ADL code you see during this tutorial, see the ADL online help. The online help also includes a description of each ADL function.

6.2 Creating a Project for Your Program

Complete the following steps to create a ADL Studio project (projects manage the files needed to run your ADL script programs from ADL Studio).

1. Start Simulated Phone and ADL Studio from the Windows Start menu.
2. On the Welcome to ADL Studio dialog, select Create a new project, then click OK to display the New dialog.

Figure 6–1 Welcome to ADL Studio Dialog



3. On the New dialog:

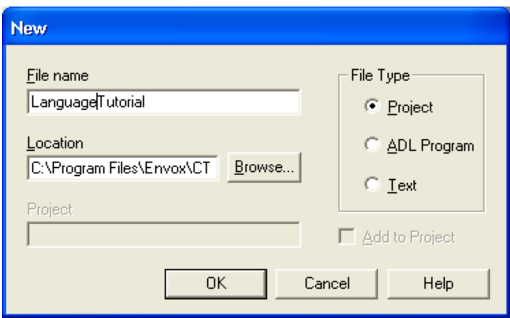
   a. Make sure that Project is the selected File Type.

   b. In the File name field, name your project **LanguageTutorial**.

   c. In the Location field, type the following path for the project:

   ```
   C:\Program Files\Envox\CT ADE\ADL\Tutorial\Language
   ```

Figure 6–2 New Dialog



   d. Click OK, and ADL Studio creates your project.

6.3 Adding a Program to the Project

Now it's time to add an ADL program to the project.

1. From the main menu in ADL Studio, click File | New.
2. In the New dialog:

   a. Make sure that ADL Program is the selected File Type.

   b. In the File name field, name the program **First**.

c. Make sure the Add to Project checkbox is checked.

d. Click OK to create the new program.

A folder called Application First is added to the project tree in the left window:

```
Project LanguageTutorial
    Application LanguageTutorial_master
    Application First
    RLLs
```

3. Double-click the Application First folder, and you'll see an icon labeled First.vs in the project tree.

   The Editor window to the right contains a skeleton ADL program.

4. Type the following lines into the Editor window between the program and end program lines:

```
vid_write("I am an ADL program!");
vid_write("Type any key to exit...");
kb_get();
vid_write("Exiting now.");
exit(0);
```

6.4 Compiling and Running the First Program

When you compile an ADL program, it gets converted from a form that you can understand (such as a source file, like First.vs) to a form that ADL can execute (a binary file—also called an executable file).

1. From the ADL Studio main menu, choose Run | Compile First.vs.

   If you didn't make any typing errors, you'll see a message telling you that your program has been compiled:

```
Compiling...
first.vs
Compilation Complete
```

   The ADL Language Compiler created a binary file called First.vx. You can use Windows Explorer to see that the binary file has been created. If you see error messages from the Compiler, check your typing by carefully comparing the file you created with the source code provided.

   Next, run the First program:

2. From the main menu, choose Run | Run First.vs.

Notice that ADL Studio compiled your program again. Any time you run a program in ADL Studio, it automatically compiles the program first. After compiling the program, ADL Studio starts the ADL runtime engine. You can see the stopwatch icon ticking in the Windows system tray, and a new window called ADL Display opens (if you have previously disabled the ADL Display, it doesn't open).

3. Right-click the ADL Display icon in the Windows task bar, and choose Restore.

   The following message appears in the ADL Display:

```
I am an ADL program!
Type any key to exit...
```

4. Type any key, and the ADL Display window closes.

   If you didn't see that message, try looking at the ADL log file. From the main menu, choose View | Runtime Log. The ADL.LOG file is created automatically by ADL. It displays a new message each time ADL is started, and each time a condition arises that may indicate an error in a program or hardware problem. If you don't find a message indicating the source of the trouble, it's time for your first tech support call. For more information, see Chapter 9.

6.4.1 Reading the Source Code

You should find the source code to be mostly self-explanatory. For example, the word *program* starts a program, and the word *endprogram* marks the end of the program.

The vid_write function asks ADL to write the given string to the ADL Display. The kb_get function waits for a keystroke, and returns the character pressed, although we ignore the returned value in this example. Most ADL programmers use languages like Visual Basic or Visual C++ to develop interfaces for ADL programs so you probably won't use functions like vid_write and kb_get very often, but they're appropriately simple for an example like this.

The exit function stops the ADL runtime engine with the given exit code (which can be tested by the IF ERRORLEVEL command in batch files, etc.).

For more information about interpreting the source code, see Chapter 5 or the ADL online help.

6.4.2 Creating a Second Call Processing Program

Now you're ready to add another ADL program to the project.

1. From the main menu in ADL Studio, click File | New.
2. In the New dialog:

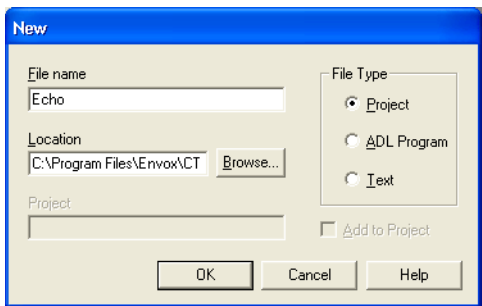
a. Make sure that ADL Program is the selected File Type.

b. In the File name field, name the program **Echo**.

c. Make sure the Add to Project checkbox is checked.

d. Click OK to create the program file.

Figure 6–3 Choosing New From the ADL Studio File Menu



3. Click on the blank line between program and endprogram in the Editor window, and then type in the following code:

```
TrunkWaitCall();
TrunkAnswerCall();
MediaRecordFile("msg.vox", 15);
MediaClearDigitBuffer();
MediaPlayFile("msg.vox");
TrunkDisconnect();
restart;
```

6.4.3 Compiling and Running the Second Program

Complete the following steps:

1. From the main menu in ADL Studio, choose Run | Run Echo.vs.

   The program automatically compiles first and then runs. Again, you can see the stopwatch icon in your system tray and the ADL Display on your tasteable.

   The program you wrote executes, and then waits for a call to come in.

2. Click Ring in the Simulated Phone main window.

   You’ll see the Line State icon in Simulated Phone go off-hook, which means that your application has answered the call.

3. Say something into your sound card’s microphone and then click any Simulated Phone digit button.

This stops the recording, and the message is played back to you. As soon as the message playback has finished, the program ends the call and starts at the beginning. You can then call in again if you want.

4. When you are finished, stop the program by choosing Run | Stop Running from the ADL Studio main menu.

This completes the tutorial. In the next section you analyze the code in the Echo Program.

6.5 Examining the Second Program

In this section, we'll dissect the Echo program you created. The following table offers a brief explanation of what happens at each line. The next section provides a more in-depth explanation.

Command	Action
`program`	Start here
`TrunkWaitCall();`	Wait for call
`TrunkAnswerCall();`	Answer call
`MediaRecordFile("msg.vox", 15);`	Record sound to file
`MediaClearDigitBuffer();`	Clear the buffer in case a caller pressed a digit
`MediaPlayFile("msg.vox");`	Play back sound
`TrunkDisconnect();`	Terminate call
`restart;`	Go back to first statement
`endprogram`	End of main program

6.5.1 Echo Program Code Detail

Every program must begin with the word *program*, and end with the word *endprogram*.

TrunkWaitCall Command

Command: `TrunkWaitCall();`

Action: Waits for an incoming call on the current trunk (phone line)

Description: This command tells ADL to suspend the program until an incoming call is received on the given line. When the call comes in, ADL “wakes up” the program, which then continues execution.

TrunkAnswerCall Command

Command: `TrunkAnswerCall();`

Action: Answers the call on the current trunk line

Description: This command instructs the telephony board to “go off hook,” meaning to answer the phone. This action corresponds to picking up the handset on a telephone.

MediaRecordFile Command

Command: `MediaRecordFile("msg.vox", 15);`

Action: Records from the current line to a hard disk file named msg.vox

Description: This command turns the telephony board into a digital tape recorder. The sound received on the phone line is digitized and saved to a file on the computer’s hard disk. Just as with MediaPlayFile, there is one mandatory parameter to this command, the hard disk file name to store the digitized sound. In addition to the file name, by setting the second parameter, you can control the maximum length of the recording, in seconds. The example shows MaxSecs set to 15, so the recording can be up to 15 seconds long.

MediaClearDigitBuffer command

Command: `MediaClearDigitBuffer();`

Action: Removes any digits from the digit buffer

Description: This command clears any digits that are stored in the digit buffer. If the caller pressed a digit on the phone to end the recording (a “stop tone”), the digit needs to be cleared from the buffer so it doesn’t also stop the next command, which plays a file.

MediaPlayFile command

Command: `MediaPlayFile("msg.vox");`

Action: Plays back sound from msg.vox on the current line

Description: This command plays back sound from a hard disk file. It has one mandatory parameter, which is the name of the file.

TrunkDisconnect command

Command: `TrunkDisconnect();`

Action: Terminates call on the current line

Description: This command finishes the call by putting the line back “on-hook” to hang up.

Restart Command

Command: `restart;`

Action: Goes back to the beginning of the program

Description: This command goes back to the beginning and starts executing again. You can think of it as jumping back to the word *program* and resuming execution from the first command.

6.6 Using Basic Telephony Functions

The next two tables list some of the main capabilities of the telephony boards. The following section describes how to write several functions you're most likely to use.

6.6.1 Audio Processing

This table lists audio processing commands:

Action	Command
Play speech from file	`MediaPlayFile`
Record speech to file	`MediaRecordFile`
Get touch tones from caller	`MediaGetDigitBuffer`

6.6.2 Telephone Signaling

This table lists telephone signaling commands:

Action	Command
Wait for incoming call	`TrunkWaitCall`
Answer incoming call	`TrunkAnswerCall`
Terminate a call	`TrunkDisconnect`
Make outbound call	`TrunkMakeCall`
Detect caller hangup	`onhangup`

6.6.3 Programming Basic Features

This section shows you the code necessary to program basic computer telephony features using ADL script.

6.6.3.1 Answering an Incoming Call

ADL answers a typical call using the following command sequence:

Action	Command
1. Wait for ring	`TrunkWaitCall();`
2. Go off-hook	`TrunkAnswerCall();`
3. Play greeting	`MediaPlayFile(filename);`

For more information and an example program that answers inbound calls, see the ADL online help.

6.6.3.2 Making an Outgoing Call

ADL usually makes an outgoing call using this command sequence:

Action	Command
1. Make outbound call	`TrunkMakeCall();`
2. Check new state	`State = TrunkGetState();`
	`Glare = TrunkGlare();`

The new state of the trunk should be Connected, NoConnect, or InboundRinging. Connected means that the call went through or that a condition called *glare* occurred, that is, an inbound call arrived just as you were attempting an outbound call. You can test for glare with the TrunkGlare function. A NoConnect state means that the call was not completed (for example, there was a busy tone, or the line rang off the hook without being picked up). InboundRinging also indicates a glare condition.

For more information and an example program that makes outbound calls, see the ADL online help.

6.6.3.3 Terminating a Call

To end a call, simply use the TrunkDisconnect function. Most programs are designed to start again at the beginning and then prepare to receive another call. You can do this using the restart command:

Action	Command
1. Hang up	`TrunkDisconnect();`
2. Return to start of program	`restart;`

6.6.3.4 Detecting a Disconnect

The caller may hang up at any point in the ADL program. Consequently, the caller might hang up when the program is not currently waiting for a hang-up or when it's executing an action.

The most commonly used mechanism in ADL is the onhangup function. When a disconnect is detected, ADL interrupts the program wherever it is currently executing, and jumps to the onhangup function. The onhangup function can contain any ADL commands. The onhangup function usually finishes with either a return command or a restart command. A return command sends the program back to the point where it was originally interrupted, and continues exactly as before (except that variables may be changed in the onhangup function). If a restart command is executed, the program goes back to the start of the program and is ready to receive the next call.

ADL Code for Creating an onhangup Function

```
onhangup
     TrunkDisconnect();
     restart;
endonhangup
```

Suitable for most ADL applications, this code reacts to a caller hang-up by disconnecting and going back to the beginning of the program, which presumably either waits for the next call to arrive or initiates a new call.

A problem with using onhangup can arise if the program is in the middle of doing something important when the disconnect signal arrives. For example, it may be making updates to a database. This situation can be handled using the TrunkDeferOnHangup and TrunkClearDeferOnHangup commands, which allow a disconnect signal to be *deferred*. Calling TrunkDeferOnHangup says, "I am beginning to do something important—don't jump to onhangup even if a disconnect signal does arrive." Calling TrunkClearDeferOnHangup says, "OK, if a disconnect signal did arrive, now jump to onhangup, otherwise just carry on as usual."

We can add disconnect processing to our example program by adding an onhangup function.

ADL Code for Adding an onhangup Function

```
program
      ADLlog("Program started");
      TrunkWaitCall();
      TrunkAnswerCall();
      sleep(10);
      MediaRecordFile("msg.vox", 15);
      MediaPlayFile("msg.vox");
      TrunkDisconnect();
      restart;
endprogram
onhangup
      TrunkDisconnect();
      restart;
endonhangup
```

6.6.3.5 Creating a Touch Tone Menu

One of the basic tools utilized in automatic call processing is the ability to play a menu and get a response from the caller. A typical audible menu pattern reads like this:

"*Please press 1 to do this, press 2 to do that, press 3 to do...*"

In order for the menu to work correctly, the pre-recorded file containing the voice saying the menu must play first. Then ADL must wait for the caller to enter a touch-tone digit.

ADL Code for Creating a Touch Tone Menu

```
MediaPlayFile("menu.vox");
MediaWaitDigits(1);
   digit = MediaGetDigitBuffer();
   switch (digit)
   case 1:
      do_one();
   case 2:
      do_two();
   case 3:
      do_three();
   default:
      bad_digit();
   endswitch
```

The MediaWaitDigits function waits for the caller to enter one or more digits. The function has one mandatory parameter, that is, the number of digits to get. Thus, MediaWaitDigits(1) waits for a single digit and moves it to the digit buffer (the place where ADL stores digits dialed by the caller).

The MediaGetDigitBuffer command returns the contents of the digit buffer.

In this example we used the commands: do_one, do_two, do_three, and bad_digit. These are *user-defined functions*, that is, new commands you can invent by writing ADL script. User-defined functions are called functions or subroutines in other languages.

7

Debugging in ADL

This chapter provides a tutorial in which you create and debug an application. It contains the following sections:

- Creating an Application
- Debugging the Application

7.1 Creating an Application

In this section, you use ADL to create an application. In the next section you debug your new application using the Debugger and Simulated Phone.

Note: The Debugger is for use with ADL only.

7.1.1 Before You Start

Before beginning this tutorial, make sure CT ADE with ADL 9.0 or higher is installed correctly. You can run this tutorial in either evaluation or development mode. You don't need a telephony card for this tutorial because you're going to use Simulated Phone.

If you have questions or want more information on any of the ADL code you see during this tutorial, see the ADL online help, which includes a description of each ADL function.

7.1.2 Creating a New Project

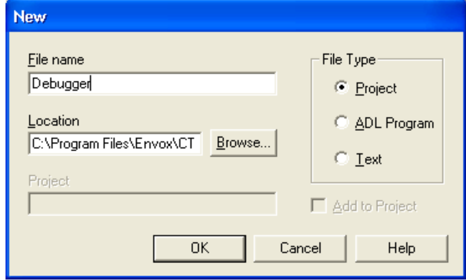
Complete the following steps:

1. Choose one of the following:
   - If you don't have ADL Studio running, launch it from the Start menu and select the Create a new project button on the Welcome to ADL Studio dialog.
   - If ADL Studio is running, close any open projects and choose New from the File menu.
2. In the New dialog (see Figure 7–1):
   a. Select Project for the File Type.
   b. Name the project **Debugger**.

c. Save it in the following directory:

```
C:\Program Files\Envox\CT ADE\ADL\Tutorial\Debugger
```

Figure 7–1 Selecting New From the ADL Studio File Menu



d. Click OK to create the project.

Now add a new ADL Program to the project:

3. From the File menu, choose New and then complete the following:

   a. In the New dialog under File Type, select ADL Program.

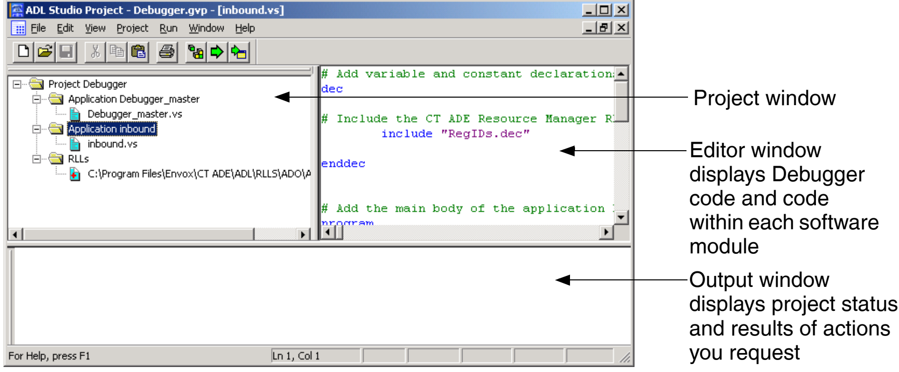
   b. Name the file **inbound**.

   c. Make sure the Add to Project box is checked.

   d. Click OK to create the file to add it to your ADL Studio project.

In the Project window, you can see a tree that shows the project's files: a master application, an RLL, and the inbound.vs application:

Figure 7–2 ADL Studio Project Window



In the Editor window to the right, you can see the skeleton file that ADL Studio has created.

The window on the bottom is the output window.

4. Click between the program and endprogram lines in the Editor window on the right and type the following code into it:

```
TrunkWaitCall();
TrunkAnswerCall();
if (TrunkGetState() strneq "Connected")
   voslog("Unexpected trunk state ", TrunkGetState());
   stop;
endif
InboundCall();
TrunkDisconnect();
restart;
```

5. Click between the onhangup and endonhangup lines in the Editor window on the right and type the following code into it:

```
if (TrunkGetState() streq "RemoteDisconnected")
   TrunkDisconnect();
endif
restart;
```

6. Click at the end of the file (or press Ctrl-End) and type the following code:

```
func InboundCall()
  MediaPlayFile("..\Prompts\Welcome.vox");
endfunc
```

7. Choose File | Save.

7.2 Debugging the Application

Now you're ready to debug the application you created.

7.2.1 Starting Debugger

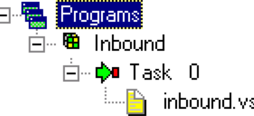
Complete the following steps:

1. In the project window, right-click the Application inbound folder and choose Debug Application.

   ADL Studio starts the Debugger and ADL. The Debugger displays in the right window—which is now called the Tasks window, and ADL displays the ADL Display dialog.

2. Minimize the ADL Display dialog.

3. If you need to, right-click on the Programs icon in the Tasks window and choose Expand Tasks Tree so that you can see Task 0.



7.2.2 Stepping Through Code

Now we can look at the Inbound program more closely.

1. In the Tasks Window, double-click Task 0 to see the source files for the task. Then double-click inbound.vs to switch to the Source File window and see the source code.

Figure 7–3 Source Code for the inbound.vs Task

```
program
   TrunkWaitCall();
   TrunkAnswerCall();
   if (TrunkGetState() strneq "Connected")
      voslog("Unexpected trunk state ", TrunkGetState());
      stop;
   endif
   InboundCall();
   TrunkDisconnect();
   restart;
endprogram

onhangup
   if (TrunkGetState() streq "RemoteDisconnected")
      TrunkDisconnect();
   endif
   restart;
endonhangup

func InboundCall()
   MediaPlayFile("HelloInbound.vox");
endfunc
```

The next statement to be executed in the task is highlighted and displayed with a “blue arrow” icon. Because we haven’t executed any lines of code yet, the next statement to execute is the first statement in the program:

```
TrunkWaitCall();
```

This waits for an incoming call. The program waits indefinitely until a call arrives.

2. Click the Step Next icon, and the highlight and blue arrow disappear. If you do not see this icon in the toolbar, choose View | Debugger Toolbar.

   ADL is executing the first line. In order for the function to return, the application must receive an incoming call.

3. On the Simulated Phone main window, click Ring to simulate a call to the voice card.

   The highlight and blue arrow move to the second line in the program:

   ```
   TrunkAnswerCall();
   ```

4. Click Step Next again and ADL answers the call.

The blue arrow and highlight move to the next line in the program.

Next we are going to execute a few lines of code without having to click Step Next repeatedly.

7.2.3 Using Breakpoints

A breakpoint is a condition that causes an ADL task to stop execution and enter a Paused state. First let's add a breakpoint to a line:

1. Click once on the line of code that reads:

   `InboundCall();`

   A different highlight, called the cursor, appears. By default, the cursor is bright green.

2. In the toolbar click the Toggle Breakpoint icon and a red hand appears next to the line.

Figure 7–4 Red Hand Indicates Debugger Stopped

```
program
   TrunkWaitCall();
   TrunkAnswerCall();
   if (TrunkGetState() streq "Connected")
      voslog("Unexpected trunk state ", TrunkGetState());
      stop;
   endif
   InboundCall();
   TrunkDisconnect();
   restart;
endprogram

onhangup
   if (TrunkGetState() streq "RemoteDisconnected")
      TrunkDisconnect();
   endif
   restart;
endonhangup

func InboundCall()
   MediaPlayFile("HelloInbound.vox");
endfunc
```

3. Next, click the Resume icon in the toolbar and the program executes up to the breakpoint.

7.2.4 Viewing the Call Stack

The next line to execute in the main program invokes a user-defined function:

`InboundCall();`

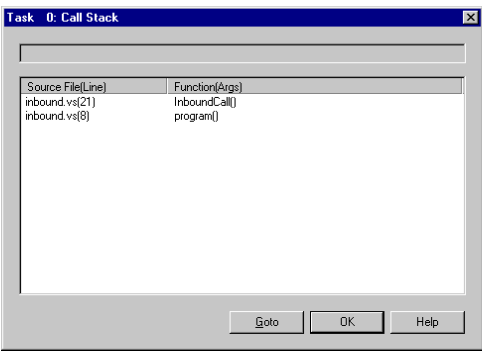
To step through this function:

1. Click the Step Into icon.

   Clicking this icon executes up to the first line of the function.

2. Next, click the Call Stack icon to display the Call Stack dialog.

Figure 7–5 Call Stack Dialog



The function call stack is a display showing the hierarchy of function calls that the program went through to get to its current position. You can see that the task is currently paused on line 21 of inbound.vs, in a function called InboundCall. That function was called from line 8 in the main program in inbound.vs.

3. Click OK to close the Call Stack window.
4. Click the Step Next icon to execute the next function, playing the Welcome.vox file.

   Finally, close the Debugger.

5. From the Run menu, click Stop Debugging.

For more information about the Debugger, see the ADL online help.

8

Troubleshooting

This chapter describes how to fix or avoid common problems you may encounter with your ADL or ADX application. It contains the following sections:

- Overview
- Troubleshooting in ADL
- Troubleshooting in ADX

8.1 Overview

There are six general steps you need to take to troubleshoot any problem you are experiencing:

1. Examine the information contained in the log file to determine which function/method or property relates to the symptom you are seeing, as well as the accompanying messages.

   The problem is almost always caused by a single function/method or property setting.
2. Localize the issue to the ADL function/ADX method or property setting that is causing the symptom.
3. Find out how to consistently reproduce the symptom.
4. If a problem cannot be reproduced, it is much harder to analyze.
5. Simplify your test project as far as possible.
6. For example, you might run the project on only one telephone line or remove all code unrelated to the symptom you are seeing, such as database accesses.
7. Check the latest information from your support provider regarding known bugs to see if there is anything related to your symptom.
8. Contact Technical Support via e-mail, or phone to report the problem.

The next two sections describe troubleshooting in more detail in both ADL and ADX.

For more information about reaching Technical Support, see Chapter 9.

8.2 Troubleshooting in ADL

At runtime, ADL creates a file named ADL.LOG that reports information about how your application is running that you may find useful when you are debugging and troubleshooting. This file is created in the same directory where your program resides, but you can specify a different directory if you want. For details on how to do this, see Section 8.2.3.

8.2.1 Viewing the Log File

You can access the log file whether or not you are in the ADL Studio. To view the log without launching ADL Studio, open any text editor, browse to find the ADL.LOG file in the application's project folder, and then open it. To view the runtime log from within the ADL Studio, open the application in ADL Studio, and then choose View | Runtime Log.

Figure 8–1 ADL.LOG File

```
ADL.log - Notepad
File  Edit  Format  View  Help
050310 110555.10 ADL loaded from C:\Program Files\Envox\CT ADE\ADL\Bin\ADLRuntime
050310 110555.10 ADL 8.3 (Debug) Built May 26 2004 15:06:59
050310 110555.10 Running as user application
050310 110555.10 Topaz loaded from C:\WINDOWS\System32\CTADE.dll
050310 110555.10 Resource Manager Built May 26 2004 15:00:40
050310 110556.82
050310 110556.82    Registered Topaz Devices
050310 110556.82    ResType  ResIndex  TechType
050310 110556.82    -------  --------  --------
050310 110556.82    Trunk      0 -   0  SimTrunk
050310 110556.82    Media      0 -   0  SimMedia
050310 110556.82    Media      1 -   1  WaveMedia
050310 110556.82      TTS      0 -   0  Sapi5TTS
050310 110556.82
Loaded UNICODE RLL C:\Program Files\Envox\CT ADE\ADL\RLLS\ADO\ADORll.dll
RLL ver=15:14:44 Address=0x01210000
        0x01224e40  _AStart,s
        0x012252e0  ABCreatePub,1
        0x01225700  ABOpPub,?
        0x01225ba0  ABCreatePrv,1
        0x01225f10  ABOpPrv,?
        0x01226380  ABUse,1
        0x01226640  ABSetAsync,0
        0x012268c0  ABClrAsync,0
        0x01226b40  ABCls,0
        0x01226ee0  ABNrTab,0
        0x01227160  ABTabNm,1
        0x01227400  ABExec,?
```

When ADL.LOG reaches the maximum size (defined in the ADL Settings file) ADL renames it ADL_First.LOG, and starts a new ADL.LOG file. After this point, all subsequent files created are named ADL_Prev.log (deleting any existing file with that name). Therefore, ADL.LOG always contains the latest information, ADL_First.log contains initial startup information, and ADL_Prev.LOG (if present) contains entries up to the start of ADL.LOG. Distributing the information in this way allows you to control the size of the log file so that it doesn't consume too much space on your system.

By default, ADL appends to any existing ADL.LOG file, and the old file is not deleted when ADL starts. To change this behavior, see Section 8.2.3.

While it may seem confusing at first, learning to use the log file can save you hours of troubleshooting on systems under development and on systems installed at customer sites. We recommend that you read this chapter carefully and take the time to review your log files regularly. The effort will repay itself many times over.

8.2.2 Interpreting the ADL.LOG Contents

This section explains the entries in the ADL.LOG file.

8.2.2.1 ADL Version

When ADL starts, an entry like the following is written to the log file:

```
000424 121445.92 ADL 8.3.0 (Debug) Built Apr 19 2004 14:23:51
```

The first field is a time stamp and it has this format: YYMMDD HHMMSS.CC (CC is hundredths of a second).

```
000424 121445.92
```

Most lines in the log file have a time stamp showing when the line was written.

The second field shows the version of ADL—version 8.3.0 in this case, and that it is running in Debug mode.

```
ADL 8.3.0 (Debug)
```

Note: In Debug mode, ADL produces more detailed and complete information in the log file. We recommend that you use Debug mode when creating your application and even when installing live systems, unless the slower performance or greater memory use of Debug mode creates problems.

The last field reflects the date and time that ADL was built.

```
Built Apr 19 2004 14:23:51
```

Seeing when ADL was built provides more exact information than the version number because the build time distinguishes between minor versions which are shipped with bug fixes and enhancements.

8.2.2.2 ADL Exit

When ADL terminates for any reason, two entries are written to the log file giving the reason why ADL is stopping and confirming that it did stop. Typical entries look like this:

```
000424 121511.11 Stopping: Command from external application
000424 121511.21 ADL stopped.
```

Unless ADL crashes catastrophically, these entries are always present after ADL terminates.

8.2.2.3 Using Built-In Function Calls

The log file entry for a typical built-in function call looks like this:

```
000426 140726.13 inbound:00c3[0] @B time() = 140726
```

The fields following the time stamp (inbound:00c3[0]) identify the .vx file, code address, and task number that made the built-in function call. In this case, the call was made from

inbound.vx at code address 00c3, task number 0. The code address is given in hex, and this is one of the few cases where information is written to the log file in hex rather than in decimal notation. The code address can be converted back to a source file name and line number using the dmpvx utility.

The @B code identifies the type of information being logged. There are a number of different @ codes, for example, @B is a built-in function and @R is an RLL function. You can use these codes in an editor or searching type of filter to quickly extract the information of interest when examining a log file. For more information, see Section 8.2.2.9.

The time() = 140726 entry shows the name of the built-in function, all the arguments passed to the function (double-quotes are not used around the string values), and the returned value from the function.

Entries display after the command has executed, but you can configure the log file to display the command before execution if you want. To do this, see Section 8.2.3.

8.2.2.4 Blocking Function Calls

Some built-in functions may block the task, in other words they do not return immediately. Instead the task is "put to sleep" (suspended) while other tasks are allowed to continue. A simple example is the sleep function, which suspends the task for a given number of tenths of a second.

ADL cannot show the returned value from a blocking function at the time the function is called so the log shows two lines: one where the function is called, and a line later on showing when the task resumes and the return value being returned at that point. This example shows how a sleep function might appear in the log file:

```
000426 144512.58 inbound:00cf[0] @B sleep(5) Task is suspending
000426 144513.08 inbound:00cp[0] @B sleep (5)
```

No return value is shown the first time the @B line is generated, since it is not yet known. The second @B line shows where the task is restarted.

The sleep function returns an empty string, so there is no = sign.

You can also see from the time stamps that sleep did indeed suspend the task for 13.08-12.58 = 5 tenths of a second.

8.2.2.5 Blocking Functions and Resource Manager State Changes

By default, ADL waits until one function requiring a Resource Manager Resource completes before executing the next function.

8.2.2.6 Example: Recognizing Blocking in Log Entries

The following log file entries were produced by invoking the MediaPlayFile function:

```
000426 145231.72 inbound:00c6[0] @B MediaPlayFile(hi.vox) Task is
suspending...
000426 145233.54 [0] @Z Media:0 Playing->Idle
000426 145233.54 inbound:00c6[0] @B MediaPlayFile(hi.vox) = 1
```

In this example, Media Resource 0 is being blocked by this function in ADL task 0: @B MediaPlayFile (hi.vox) Task is suspending...

An @Z entry shows that a Resource Manager Resource State is changing. In this case Media Resource 0 is changing from the Playing state to the Idle state. (For more on Resource Manager Resource states, see the ADL online help.)

The MediaPlayFile(hi.vox) = 1 entry shows that the task has been “woken up” again. In this example, task 0 was resumed and the return value from MediaPlayFile is 1.

8.2.2.7 Asynchronous Mode

Asynchronous mode lets you start one function that requires a Resource Manager Resource and then continues your program’s execution before that function finishes. When you’re running in asynchronous mode, functions that don’t use the Resource Manager Resource can appear between the function that blocked the Resource and the notification that the Resource was blocked.

8.2.2.8 Example: Recognizing Blocking in Log Entries in Asynchronous Mode

Time() function executes while the MediaPlayFile function is playing:

```
000426 152447.56 inbound:00b2[0] @B MediaEnableAsync() = 1
000426 152447.59 inbound:00ce[0] @B MediaPlayFile(hi.vox) = 1
000426 152447.59 inbound:00d5[0] @B time() = 152447
000426 152447.59 inbound:00df[0] @B MediaWait() Task is
suspending...
000426 152449.40 [0] @Z Media:0 Playing->Idle
000426 152449.40 inbound:00df[0] @B MediaWait() = 1
```

8.2.2.9 Interpreting @ Codes in the Log File

The codes in Table 8–1 are used as prefixes to various log file entries. These codes are designed to allow a text editor or text search program to identify relevant lines in ADL.LOG.

Table 8–1 Prefix Codes in Log File Entries

Code	Meaning
@b	Built-in function call and the arguments used before the call is made. Tracing before and after function calls must be enabled to see @b codes
@B	Built-in function call and result. At this point, the call has already been made

Table 8–1 Prefix Codes in Log File Entries

Code	Description
@D	Intel Dialogic API call or event
@E	User error (voslog call starting with "@E...)
@F	Built-in function call failure message
@L	User log message to file only (adllog "@L...)
@N	Resource Manager Notify (a resource type- or Technology-specific notification, such as a digit detected event)
@O	Intel Dialogic API call "on entry," before calling the actual function (API)
@Q	Resource Manager Operation Complete (completion event for a blocking Get/Set or [Un]Listen)
@r	RLL function call and the arguments used before the call is made. Tracing before and after function calls must be enabled to see @r codes
@R	RLL function when function returns
@S	Data Structure. This code dumps a C struct or similar passed in or out as a function argument
@V	Assignment to variable (watchvar)
@W	Built-in function warning
@X	Failed Intel Dialogic API call or event
@Y	Technology-specific event
@Z	Resource Manager state change

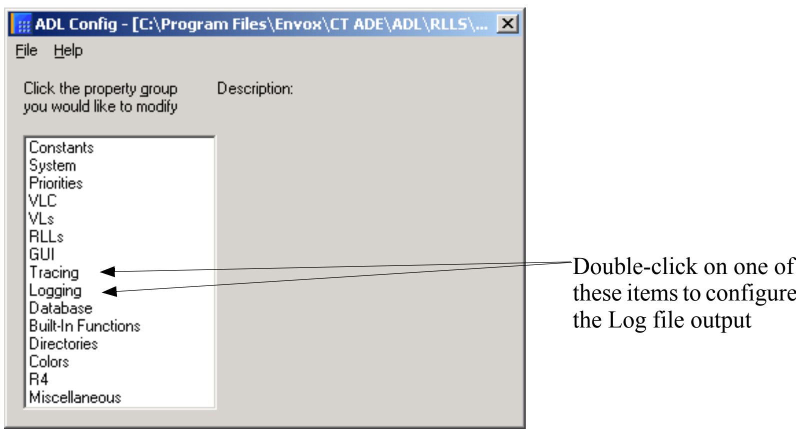
8.2.3 Configuring the ADL.LOG File

You can determine the output of the log file using two dialogs from the ADL Config dialog list: the Logging dialog and the Tracing dialog.

1. Launch ADL Studio and open a project.
2. Choose Project | ADL Settings.

   The ADL Config dialog displays:

Figure 8–2 ADL Configuration Dialog

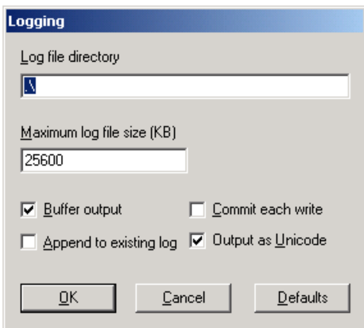


3. Choose one of the following:
   - To access the Logging dialog, see Section 8.2.3.1.
   - To access the Tracing dialog, see Section 8.2.3.2.

8.2.3.1 Configuring the Logging Dialog

1. From the ADL Config dialog, double-click Logging from the list.

Figure 8–3 Logging Dialog



2. From this dialog, change the following log file settings as appropriate:

Entry	Meaning
Log file directory	Directory in which the ADLx.LOG files are stored
Maximum log file size (KB)	Sets the maximum size of the ADL log. Default is 25,600 Kb
Buffer output	Check this box to enable buffering or uncheck to disable buffering. If Buffered, ADL keeps > 1 line of log output in memory and writes a group of lines at one time. If not buffered, ADL writes each line as it is generated
Append to existing log	Check this box to append new entries or uncheck to overwrite any old log when you start ADL
Commit each write	Check this box to commit the log file to disk each time a line is written. (Committing is the equivalent of asking Windows not to buffer.) Committing is required only if Windows doesn't complete a write in the case of a crashed process. This is usually not necessary because Windows will write any buffered output to the file even if a process crashes. Since this will be much slower, leave this box unchecked unless really needed.
Output as Unicode	Check this box to output the ADLx.LOG file in Unicode or uncheck the box to output the log in ASCII. The default value is checked.

3. When you're finished, click OK to close the dialog.

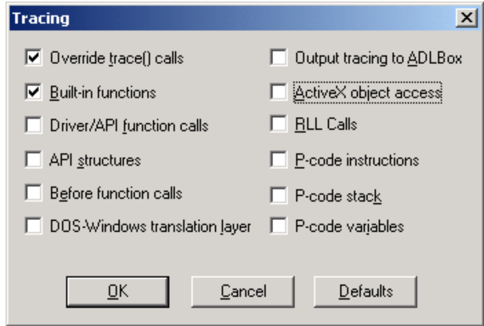
   Your changes go into effect the next time you run your application.

8.2.3.2 Configuring the Tracing Dialog

1. From the ADL Config dialog, double-click Tracing from the list.

   The Tracing dialog displays.

Figure 8–4 Tracing Dialog



2. From this dialog, change the following trace settings as appropriate:

Entry	Meaning
ActiveX object access	Check this box to enable ADL to provide detailed tracing of object accesses (available in Debug mode only), or uncheck to disable it
Built-in functions	Check this box to trace all ADL built-in functions, or uncheck to disable tracing of built-ins
Drivers/API function calls	Trace API function calls
Before function calls	Trace both before and after function calls. This is useful when a function call hard-crashes ADL. Normally functions are logged only after they return, but if the function crashes this doesn't show in the log.
DOS-Windows translation Layer	Log DOS-to-Windows translation layer for 'legacy' functions sc_, DTI_ etc.
Output tracing to ADL Box	Check this box to display the tracing information in the ADL Display and write it to the log file, or uncheck to just write it to the log file
Override trace() calls	Check this box to override tracing options set with the trace() function in individual programs, or uncheck to use the program's settings

P-code instructions	Check this box to trace all p-code instructions or uncheck to disable p-code tracing (P-code tracing creates a large amount of logging)
RLL Calls	Check this box to trace all RLL calls or uncheck to disable tracing of RLLs
P-code stack	Check this box to include the stack in p-code tracing or uncheck to omit it
API structures	Check this box to trace API structure members or uncheck to omit structures
P-code variables	Check this box to include variables in p-code tracing or uncheck to omit them

3. When you're finished modifying the log file settings, click OK to close this dialog.

   Your changes go into effect the next time you run your application.

8.3 Troubleshooting in ADX

ADX components have the ability to keep a log of activities on the telephony channel. This log file, called ADX.log, can help you understand how the ADX application is running as well as help you troubleshoot problems you encounter. When your application behaves in an unexpected way, search the ADX.log file for any errors, warning messages, events, or other information that may help you pinpoint the problem.

When you run an application in ADX, the ADX.log file is created here:

```
C:\Program Files\Envox\CT ADE\Common\Log
```

Figure 8–5 ADX.log File

```
ADX.log - Notepad
File  Edit  Format  View  Help
14:27:42.02                    Log Server Started on 2005/03/10 14:27:42.
14:27:42.02                    LogServer.exe loaded from C:\Program Files\Envox\CT A
14:27:42.02                    LogServer.exe Built May 26 2004 15:04:07
14:27:42.02                    ADX.INI Settings
14:27:42.02                         LogDir  =C:\Program Files\Envox\CT ADE\Common\Lo
14:27:42.02                         LogForce=No
14:27:42.02                         LogStartup=Yes
14:27:42.02                         LogFileAppend=No
14:27:42.02                         LogUnicode=Yes
14:27:42.02                         MaxLogfileSize=25000000
14:27:42.02                         ErrorPopup=Yes
14:27:42.23                    Log client connected
14:27:42.23             SPhone @P Log=1
14:27:42.23             SPhone @P ANI=""
14:27:42.23             SPhone @P DNIS=""
14:27:42.23             SPhone @P CallerName=""
14:27:44.42                    Log client connected
14:27:44.42 P000T000 V[i00] Voice Object Built May 26 2004 15:21:45 loaded from C
14:27:44.42 P000T000 V[i00] Envox CT ADE Voice Object v8.3
14:27:44.49                    Module run mode init
14:27:44.49 P000T000 V[i00] ModuleRunModeInit of ADXVoice: Resource Manager Built
14:27:44.49 P000T000 V[i00] Run mode init
14:27:44.49 P000T000 V[i00] RunModeInit of ADXVoice: Resource Manager Built May 2
14:27:44.54 P000T000 V[i00] @W ADXVoice loaded from C:\Program Files\Envox\CT ADE
14:27:44.54 P000T000 V[i00]    Trunk     0 -   0  SimTrunk
14:27:44.54 P000T000 V[i00]    Media     0 -   0  SimMedia
14:27:44.54 P000T000 V[i00]    Media     1 -   1  WaveMedia
14:27:44.54 P000T000 V[i00]      TTS     0 -   0  Sapi5TTS
```

When ADX.log reaches the maximum size (defined in the ADX.ini file), it is renamed ADX_First.log, and starts a new ADX.log file. After this point, all subsequent files created are named ADX_Prev.log (deleting any existing file with that name), and a new ADX.log file is started. This prohibits the log file from growing without limits and filling up your disk. This also means that ADX.log always contains the latest information, ADX_First.log contains initial startup information, and ADX_Prev.log (if present) contains logs up to the start of ADX.log. By default, ADX appends to any existing ADX log file, and the old file is not deleted when ADX starts.

When you first see the ADX log file, it may look difficult to understand. However, we encourage you to review the log often and use the log entry descriptions provided here (and in the ADX online help) to become more familiar with it. As you become more experienced, you will find that it is a valuable resource.

8.3.1 Interpreting the ADX.log File Contents

Depending on your settings via the SetLogLevel methods and INI file entries, the ADX.log file contains useful records of what is happening as your application executes. Typical entries include:

- Methods and method parameters
- Telephony driver function calls
- Telephony events and messages
- Information about the runtime environment, such as the versions of DLLs used by the ADX components

There are two levels of logging: standard and detailed. Many entries appear in both standard and detailed although detailed logging contains additional entries.

Each line in the ADX.log file has the following format:

```
HHMMSS.hh PpppTttt X[#] @c Text
```

The following log entries appear in both standard and detailed logging:

Log Entry	Description
`HHMMSS.hh`	Time in Hours, Minutes, Seconds and hundredths of a second when the log entry was made
`ppp`	Process index of the process that made the log entry
`ttt`	Thread index of the thread that made the log entry
	Process index and Thread index are zero-based counters that indicate which thread or process made the log entry. For example, in ADX Voice Object, if you are running 24 lines in 24 .EXEs there will be 24 processes, each with 2 threads: one thread for the main application and one thread for the Intel Dialogic drivers. The Process indexes will be numbered from 0 to 23 and the Thread indexes will be from 0 to 47.
`X`	ADX component that made the log entry and is one of the following: V = ADX Voice Object, T = ADX TTS Object, C = ADX Conference Object, F = ADX Fax Object, A = ADX ASR Object
`#`	Index number of Resource that made the log entry. If a Resource has not yet been assigned to the component (for example, during startup), the instance number of the ADX component displays instead.
`@c`	Code that indicates the log entry type. If the @c is not present, the entry is informational.
`blank`	The text is informational
`@E`	An error has occurred. Logged for all levels of logging (even if logging is disabled)
`@L`	User logging—your application has called the LogWrite method
`@M`	The ADX component is exiting a method
`@N`	A Resource Manager notify event has occurred
`@P`	The ADX component is exiting a property get or set
`@Q`	A Resource Manager Operation has completed
`@V`	An ActiveX event (often a Simulated Phone event) has occurred

`@W`	An ADX component has loaded
`@Y`	A Resource Manager event has occurred
`@Z`	A Resource Manager Resource's state has changed

These entries appear in detailed logging only:

Log Entry	Description
`@m`	The ADX component is entering a method
`@p set`	The ADX component is entering a property set
`@P get`	The ADX component is entering a property get

Each time an instance of an ADX component loads, you can see a confirmation of the path name to the control that loaded:

```
14:48:14.29 V[0x3cf6d70] @W ADX Voice Object loaded from C:\Program
Files\Envox\CT ADE\ADX\Bin\ADXVoice.dll
```

The exact date and time of the control build is useful information to Technical Support. It can be used to determine if known bugs or corrections made through point releases are present in the version of the control that you are using:

```
14:05:44.88 V[0x3cf6d70] ModuleRunModeInit of ADX Voice Object:
Resource Manager Built May 18 2001 09:40:25
```

Methods are logged, together with their parameters:

```
14:05:48.82 V[i01] @M PlayFile FileName="hello.vox" SoundFileType=0
StopTones="0123456789#*" SkipSecs=0 LengthSecs=0
```

8.3.2 Controlling Log Output

There are a number of ways you can modify the output that ADX controls send to the log file. The primary approach is to use one of these SetLogLevel methods along with a log value that determines the degree of information you see for a particular ADX component.

SetLogLevel Methods	Description
`Call ADXVoiceObject1.SetLogLevel(LogLevel)`	Sets the current log level for ADX Voice Object
`Call ADXTTSObject1.SetLogLevel(LogLevel)`	Sets the current log level for ADX TTS Object

`Call ADXConferenceObject1.` `SetLogLevel(LogLevel)`	Sets the current log level for ADX Conference Object
`Call ADXFaxObject1.` `SetLogLevel(LogLevel)`	Sets the current level for ADX Fax Object
`Call ADXASRObject1.` `SetLogLevel(LogLevel)`	Sets the current level for ADX ASR Object

Use one of the preceding methods along with one of the following three log levels to set the level of detail you want the log file to show for that component. For example:

```
Call ADXASRObject1.SetLogLevel(LOG_Enabled)
```

Log Value	Value	Meaning
LOG_Disabled	0	No log output
LOG_Enabled	1	Normal log output (default)
LOG_Detailed	2	Maximum log output (required for logs sent to Technical Support)

Additionally, you can further alter log output by modifying the settings in the [Control] section of the ADX.ini file, which is located in C:\WINNT.

Log File Setting	Valid Values	Default	Description
`LogDir`	A valid DOS/Windows drive and/or directory name.	Write log file in current working directory.	Specifies the directory where the ADX.log file should be written.
`LogForce`	Yes, No	No	Forces detailed logging when LogForce is set to Yes even if the Log level is set to LOG_Disabled. This is convenient for troubleshooting at a customer site where you have a problem report. You can obtain log output without having to modify your application.
`LogStartup`	Yes, No	No	Logs parameters which affect the initialization process if set to Yes. For example, the Intel Dialogic API calls made to initialize the telephony card together with .INI file entries are logged.
`LogBuffer`	Yes, No	Yes	Not currently used

Log File Setting	Valid Values	Default	Description
`ErrorPopup`	Yes, No	Yes	Displays an error message using a popup window when ADX receives an error. Setting this parameter to Yes stops application, so you may find it more useful during debugging than at production time
`LogFileAppend`	Yes, No	No	Appends to an existing log file when the first ADX Object is started if set to Yes. If set to No, the first ADX Object deletes any existing log file from previous runs before starting.
`MaxLogFileSize`	1024 to 2000000000	25,000,000 (25 Mb)	Specifies the maximum size of the log file. If the file becomes larger than what you specified, then the current log file is renamed ADX _Prev.log—overwriting any existing file with this name, and a new log file is started.

9

Technical Support

This chapter describes the steps to take before you contact Technical Support to ensure fast resolution to your problem.

- Overview
- What Support Needs From You

9.1 Overview

Product issues need to be reported to the support organization of the company from whom you purchased the CT ADE software. This could be a reseller, systems integrator, or Envox. The method of contact, e-mail or phone, can vary by vendor and should be explained to you by your vendor. Please contact your vendor, or refer to your support agreement, for further information.

We strongly suggest you use e-mail, whenever possible, to achieve the fastest turnaround. When you send an e-mail, your issue is ready for an engineer to investigate it. Additionally, using e-mail facilitates the ability to copy and edit details about your issue as it gets distributed internally within the company. For example, a support engineer can forward part of your message to a development engineer, or a reseller can forward your message to Envox, if needed.

While telephone support may be offered, there is no guarantee that an engineer will be available immediately to answer a question, so it may be necessary for an engineer to call you back.

9.2 What Support Needs From You

This section lists the information needed from you concerning your technical support issue. If you are calling Technical Support, make sure you have this information readily available. If you are sending an e-mail, you can attach the information in a ZIP file.

1. Specify which build of the product you are using by recording the creation date written to the log file.
2. Note which development environment you are using (ADL or ADX).
3. Specify the operating system and version you are using.
4. List the telephony hardware and driver configuration (and their version numbers) installed on your system.
5. Describe the problem, including the steps to take in order to reproduce it. It is also useful if you can provide a simple test project that shows the problem (please do not include your full application unless it is very simple).
6. Include the ProfileConfig.txt file (from ...*\Envox\CT ADE\Common\ Config\Bin*).
7. Include the log file produced when you run the application:
   - For ADL, include the ADL.LOG file produced when you run the application (or include the ADL_First.LOG or ADL_Prev.LOG if the problem shows up in that file). Use detailed logging when you generate the file and make sure to include at least the built-ins and drivers in the detail.
   - For ADX, include the ADX.log file (or ADX_First.log or ADX_Prev.log file if the problem shows up in that file). Use detailed logging when you generate the log file, either by using the SetLogLevel methods with the LogLevel parameter set to LOG_Detailed or by setting LogForce=Yes in the ADX.ini file.

     For more information about important log file entries and how to control the log output, see Chapter 8.
8. If your issue is especially urgent, make this clear in your report and the issue can be looked into more quickly.

A

Glossary

A&B Bits

A&B bits are used in digital environments to convey signaling information. A bit equal to “one” generally corresponds to loop current flowing in an analog environment, a bit equal to “zero” corresponds to no loop current—that is, to no connection. Other signals are made by changing bit values. For example, a flash-hook is sent by briefly setting the A bit to zero.

ABCD Bits

ABCD Bits are signaling bits that are used on E-1 digital trunks to convey information about the state of a call. For example, ABCD bit values may be used to signal an incoming call, disconnect, seize, and so on. They are closely analogous to the A and B bits commonly used on T-1 digital trunks. E-1 is a digital trunk standard used in many countries outside of North America. There are 32 channels (time-slots) in contrast to 24 channels on T-1, hence the faster bit rate of 2.048 MHz versus 1.544 MHz for T-1. Channels 0 and 16 are used to carry the ABCD bits and synchronization (framing) bits, hence only 30 channels are available for audio conversations. Thus, ABCD bits are carried out-of-band in contrast to the in-band robbed-bit scheme used by T-1.

ACD (see Automatic Call Distributor)

ActiveX

ActiveX controls use COM technologies to provide interoperability with other types of COM components and services. An ActiveX control is an example of an ActiveX object. ActiveX objects are units of executable code (usually stored in an EXE, DLL or OCX file) and data. Usually code is shared by all objects of a given type, called a class, but a separate block of data is created for each object. ActiveX servers provide ActiveX objects.

An application that uses an object is called an ActiveX client. Visual Basic is a popular example of an ActiveX client. You can easily load and use ActiveX objects such as ActiveX controls (also called custom controls) from within Visual Basic.

AD

APPLICATION Development

AD ActiveX Objects (ADX)

The ADX implementation of the CT Application Development Environment is a collection of ActiveX components that let you add telephony features to your existing systems or create applications from scratch. ADX custom components can be used in any Windows development environment that supports ActiveX components.

AD Communication Hub (communication technology)

Available both as an ActiveX control and as an ADL RLL, AD Communication Hub is an enterprise component that supports communication (messaging) between threads in a single process, between processes in a single PC, and/or between nodes on a local or wide-area network.

AD Language (see ADL)

Adaptive Differential Pulse Code Modulation (ADPCM)

ADPCM is a speech coding method that calculates the difference between two consecutive speech samples in standard PCM-coded telecom voice signals. This calculation is encoded using an adaptive filter so it is transmitted at a lower rate than the standard 64 Khbps technique. Usually ADPCM allows an analog voice conversation to be carried within a 32k-kbit/s digital channel; 3 or 4 bits are used to describe each sample, which represents the difference between two adjacent samples. Sampling is typically done 8,000 time a second (8 KHz). The Dialogic default ADPCM format, however, uses 4-bit sampling at 6 KHz.

ADL

Application Development Language. The ADL portion of CT ADE is a complete development environment that includes the Editor, compiler, runtime engine, Debugger, and Simulated Phone.

You can create and deploy telephony systems with ADL, a complete development environment that includes the Resource Manager, the ADL Compiler, and runtime engine, Editor, Runtime Link Libraries, Debugger, and Simulated Phone. ADL provides an environment in which you can write, test, debug, and run your CT applications in a Windows environment. The ADL engine is responsible for executing the applications built with the ADL Compiler.

ADL Compiler

The ADL Compiler translates ADL source code into an ADL executable, which can then be used by the ADL runtime engine.

ADX (see AD ActiveX Objects)

A-law Encoding (also see Mu-law Encoding)

A-law encoding refers to the assignment of a “loudness” value to an audio signal using an A-law translation table. Digital telephony equipment usually stores and transmits audio

signal in 8-bit samples using PCM (Pulse Code Modulation) encoding. The value of the 8-bit sample corresponds to the amplitude (volume or loudness) of the audio signal at the time the sample was taken. A translation table is used to determine the amplitude given the sample value. The two tables in common use are the A-law table, which is widely used in Europe and Asia on E-1 trunks, and the Mu-law (m-law) table, which is used in the US, Canada, and several other countries.

Amplitude

The amplitude of a waveform or a signal refers to the distance between the high and low point of the wave. This determines the loudness or volume of the waveform or signal. Also known as wave height.

Analog (also see Digital)

Analog is derived from the word “analogous.” In telephone transmission, the signal that is being transmitted—voice, video, or image— is “analogous” to the original signal. For example, if you speak into a microphone and see your voice on an oscilloscope, the two signals would look essentially the same. The only difference is that the electrically transmitted signal (the one over the phone line) is at a higher frequency. In telecommunications, analog means telephone transmission and/or switching that is not digital. Outside the telecom industry, analog is often called linear and covers the physical world of time, temperature, pressure, and sound, which are represented by time-variant electrical characteristics, such as frequency and voltage.

ANI (Automatic Number Identification)

Also known as Caller ID, ANI is a telephony feature that allows the receiver of a phone call to see who’s calling before picking up the incoming call. The digits of the caller can arrive in analog or digital form. They may arrive as touchtone digits inside the phone call or in a digital form on the same circuit or on a separate circuit. The person receiving the call needs appropriate equipment in order to decipher the digits and to do something with them, such as add them to a database and display the matching client record on a screen so the receiver has access to the client’s information in order to provide faster customer service.

Applications Programming Interface (API)

An API is a set of instructions (called functions) provided by an operating system or device driver. Usually the functions are provided as subroutine calls that can be used by programs written in C or another programming language. An API gives users an interface to interact with, and from this interface the user can initiate contact with the underlying network services, telephone equipment (and in the case of CT ADE—Resource Manager) to request that low-level services and tasks be performed.

ARU (see Audio Response Unit)

ASR

Automatic Speech Recognition

Asynchronous Mode

Asynchronous mode is a mode for calling driver functions in which a function call to perform a time-consuming action like playing a file returns immediately to the application once the operation is started. Later, a message or event is sent that notifies the application when the operation is complete.

Audio

Audio refers to the sound you hear over the telephone. This sound has been converted to electrical signal for transmission.

Audio Frequencies

Audio frequencies refer to a range of frequencies that the human ear can hear (usually between 15 and 20,000 hertz). Frequencies transmitted through the phone range from 30 to 3,000 hertz, which is why the phone doesn't sound "Hi-Fi."

Audio Menu

An audio menu is the set of options spoken by a voice processing system. The caller chooses a menu option by pressing a touchtone on the phone or speaking a word or two. Computer or voice processing software can be categorized as menu-driven or non-menu driven programs. Menu-driven programs are easier to use, but they can only present as many options as can reasonably be spoken in a few seconds. Audio menus are typically played to callers in automated attendant/voice messaging, voice response, and transaction processing applications.

Audio Messaging Interchange Specification (AMIS)

AMIS is series of standards developed t address the problem of how voice messaging systems produced by different vendors can network or inter-network. Before AMIS, systems from different vendors could not exchange voice messages. AMIS deals only with the interaction between two systems for the purpose of exchanging voice messages. It does not describe the user interface to a voice messaging system, specify how to implement AMIS in a particular system, or limit the features a vendor may implement.

Audio Response Unit (ARU)

An ARU device translates computer output into spoken voice. That is, it provides synthesized voice responses to dual-tone, multi-frequency (DTMF) signaling input. These devices process calls based on callers input, information received from a host database, and

information carried with the incoming call (for example, the time of day). ARUs are used to increase the number of information calls handled and to provide consistent quality in information retrieval. For example, let's say you dial a computer and you hear, *"If you want the weather in Chicago, press 123."* You press the 1, 2, and 3 digits and then you hear the weather report for Chicago. The weather report was "spoken" by an ARU.

Audiotex (also Audiotext)

Audiotex is a generic term for interactive voice response equipment and services. Audiotex is to voice what online data processing is to data terminals. The idea is that you call a phone number and a machine answers, presenting you with several options such as, *"For information on plays, press 1, for information on movies, press 2,"* and so on.

Audiotst

Audiotst is an application that determines the number of voice boards present on your system. Then it plays a set of Wave files in various formats to each board to find out if those formats are supported. If they aren't, Audiotst reports an error message. You can use Audiotst with both ADL and ADX.

Automated Attendant

An automated attendant is a device that is connected to a PBX. When a call comes in, this device answers it and plays a greeting such as, *"Thanks for calling ABC Company. If you know the extension you'd like, press that number now and you'll be transferred. If you don't know it, press 0 and the live operator will assist you."* Sometimes the automated attendant might give you other options, such as, "dial 3 for a directory." Some people react well to automated attendants, but others do not so it's always a good idea to let people know that the operator is automated.

Automatic Call Distributor (ACD)

ACD is a specialized phone system used for handling many incoming calls. Once used only by airlines, rent-a-car companies, hotels, etc., it's now used by any company that has a large number of incoming calls. An ACD performs four functions: recognizing and answering an incoming call, referencing its database to find out how to handle the call, routing the call to the appropriate place, and finally, routing the call to an operator.

Automatic Call Sequencer (ACS)

ACS is a device that handles incoming calls. Typically it performs three functions. First, it answers an incoming call, plays a message to the caller, and then puts the caller on hold. Second, it signals the person taking the call which line to pick up. Finally, it provides management information such as how many abandoned calls there were, how long the longest person was kept on hold, how long the average wait was, etc.

Automatic Call Unit (ACU)

An ACU is a device that places a telephone call on behalf of a computer. ACUs are often used by telemarketing companies.

Automatic Callback

Automatic Callback describes a telephony feature in which a caller requests a "callback." For example, if a caller dials another internal extension and finds it busy, he can dial specific digits on his phone or press a special "automatic callback" button. When the person he tried to call ends her call, the phone system rings her number and the number of the original caller and then the phone system automatically connects the two lines. This feature saves time by automatically retrying the call until the extension is free.

B-Channel (also see D-Channel)

A B-Channel is an ISDN channel use to convey audio or data rather than signaling information. B stands for "bearer", as in the bearer of glad tidings.

Board Device

In the Dialogic API, some function calls reference board devices in order to set parameters or to perform operations that affect all channels on that board. A board device may be a "virtual board," that is, a single physical board may include more than one virtual board and hence more than one board device.

Cadence

In voice processing, cadence describes the pattern of tones and silence intervals generated by a given audio signal. Examples are busy and ringing tones. A typical cadence pattern is the US ringing tone, which is one second of tone followed by three seconds of silence.

Call Center (also see Predictive Dialing)

A call center is a place where calls are made and answered. Call centers usually have lots of agents, an automatic call distributor (or automatic call unit), a computer for order-entry, and lookup on customer orders. A call center may also have a predictive dialer for making lots of calls quickly.

Call Completion

Call completion indicates that a call has "gone through." When a call has been completed, there is an unbroken (complete) circuit made between the caller and the recipient of the call. This circuit is known as the talk path.

Call Progress Analysis (CPA)

CPA is the automated determination by a piece of telecommunications equipment as to the end result of dialing a number. For example, the result of the analysis might be a busy tone, ringing at the other end but no answer after a pre-set number of rings, or an answered call.

The analysis also involves detecting various call progress tones which are generated by the telephone network as the call is put through.

Call Progress Monitoring (also see Call Progress Analysis)

Closely related to call progress analysis, call progress monitoring must be active during the entire length of a conversation, not just as a number is being dialed or while a transfer is taking place. For example, when a call is placed across a PBX or to a country that does not provide for loop current drop disconnect supervision, the equipment listens for a “re-order” or dial tone to determine that the caller hung up.

Call Progress Tone

Call progress tones are sent from the telephone switch to inform the caller of the status of the call. Examples are the dial tone, busy tone, ringback tone, and error tone.

Call Supervision (also see Call Progress Analysis and Call Progress Monitoring)

Call supervision is a general term describing either call progress analysis or call progress monitoring.

Caller ID (see ANI)

Calling Tone

A calling tone is the “whistle” tone (1,100 Hz) that a fax machine makes to inform the caller that it is ready to receive a transmission.

Central Office (CO)

In North America, a CO is a location that houses a switch to serve local telephone subscribers. Sometimes the words central office are confused with the switch itself. Outside the US, a CO is more commonly known as a public exchange.

Channel

A path of communication, either electrical or electromagnetic, between two or more points. Also called a circuit, facility, line, link, or path.

Channel Device

In the Dialogic API, most function calls address channel devices. A channel device typically processes the audio from a single telephone connection.

Co-Articulation

When speaking a pair of words such as “seven nine” or test tube,” people generally omit the consonant that starts the second word, saying something like, “seven’ine” or “test’ube.” This phenomenon is called co-articulation.

Conference

A conference is a configuration where three or more callers are able to talk to each other. Some parties may be in a “listen-only” mode where they can hear other parties but cannot speak to them.

Continuous Speech

Continuous speech is a telecom term used to describe speech that is made up of a series of utterances made in relatively quick succession with co-articulation. Continuous speech is usually applied to the capability of a voice recognizer to recognize words from this type of speech.

CT ADE

Computer Telephony (CT) Application Development Environment (ADE) is a set of development tools and programming interfaces that help shorten both your time to market and your time to revenue by making it quick and easy to build robust, portable CT applications. The underlying Resource Manager architecture eliminates the need for developers to take time learning current and new hardware and API protocols. The Resource Manager architecture is an abstraction layer that sits on top of an API and performs low-level CT tasks, letting you focus on building your applications without worrying about these low-level operations. You can access the Resource Manager using either of the two included programming environments: ADX or ADL.

CT Connect

Envox CT Connect is computer telephony (CT) call control server software capable of connecting a wide range of telephone switches to a variety of data processing environments. The software's client/server technology supports industry-standard hardware, operating systems, network services, and call control programming interfaces such as C, C++, Java, TAPI, and ActiveX, letting application developers easily integrate more intelligent call control features into their existing business applications.

CTADE.ini file

The CTADE.ini file is used to specify various configuration parameters (such as which Telephony Technologies to exclude from the Resource Manager Profile scan) that the Resource Manager reads each time it is invoked on your system. Before you change any entries in the CTADE.ini file, you must close all instances of ADL or ADX and any CT ADE utilities that use the Resource Manager (such as ProfileConfig.exe or Phraser.exe) so that the changes can take effect.

D-Channel (also see B-Channel)

D-Channel is the data channel on an ISDN trunk. On a Primary Rate Interface ISDN trunk, there are 24 channels (time-slots) just as on a usual T-1 trunk. One or more channels (called D-channels) are used to convey information such as dialed digits, ANI and DNIS

information, routing, and billing codes, depending on the type of ISDN service used. Therefore there are only 23 or fewer audio channels (called B-channels) available. This is a form of out-of-band signaling protocol that can give faster responses and more flexible services than the in-band DTMF and robbed-bit technology in wide use today.

Debugger

The Debugger is a source-code level graphical debugger that lets you test and troubleshoot ADL programs through the development phase and after installation and deployment. You can use this tool to debug many programs in a single session, check program variables, and check the current state of your application without interrupting the calls in progress. Debugger is for use with ADL only.

Digital (also see Analog)

In telecommunications, recording, or computing, digital refers to the use of binary code to represent information (see Pulse Code Modulation). Alternatively, analog signals like voice or music are encoded digitally by sampling the voice or music analog signal many times a second and assigning a number to each sample.

Recording or transmitting information digitally has two major benefits. First, the signal can be reproduced exactly. This is important because in a long telecommunications transmission circuit the signal progressively loses its strength and begins picking up distortions, static, and other electrical interference "noises." In analog transmission, the signal, along with all the garbage it picked up, is simply amplified. In digital transmission however, the signal is regenerated (that is—squared off) to what it was identically. It's put through a little "Yes-No" question. Is this signal a "one" or a "zero?" Then the new signal is amplified and sent along its way. This makes digital transmission much "cleaner" than analog transmission. The second major benefit of digital is that the electronic circuitry that handles digital is becoming less expansive and more powerful.

Digital Trunk

A digital trunk is a generic term for a telephone connection that uses digital rather than analog transmission technology. Common examples are T-1 in the US and E-1 in Europe.

Digitization

Digitization is the process of converting an analog signal to a digital signal by measuring the value of the analog signal at regular intervals and converting this to a numerical value (called a sample).

DLL (see Dynamic Link Library)

DNIS (Dialed Number Identification Service)

DNIS is a feature of 800 and 900 lines that tells the receiver of the call which number the caller dialed. For example, let's say that you subscribe to several 800 numbers. You use one line for testing your advertisements on TV stations in Phoenix, another line for testing your

advertisements on TV stations in Chicago, and a third line for Milwaukee. Next you get an automatic call distributor and you terminate all the lines in one group to your ACD because it's cheaper and more efficient to manage this single group of incoming lines. You ask all of your agents to answer all of the calls, regardless of where they originate. However you need to know where each of the calls are coming from, so you ask your long distance carrier send you the DNIS for each of the calls—this is the number each person dialed to reach you. Depending on the technical arrangement you have with your long distance carrier, the DNIS digits may come to you in-band or out-of-band, ISDN or data channel, etc.

Dongle (see Hardware key)

DTMF (Dual Tone Multi-Frequency)

DTMF is another term for touchtone dialing. When you dial a digit, a tone is produced. This tone, also called a signaling digit, is actually a combination of one high-frequency tone and one low-frequency tone. Two tones are used together to signify a digit instead of just one in order to prevent the possibility of a human voice being mistaken for a tone.

DTMF Cut-Through

DTMF cut-through is the ability of CT equipment to respond immediately to a received touchtone even when a voice prompt is being played.

Dynamic Link Library (DLL) (also see Runtime Link Library)

A DLL is a library that gets linked to application programs when they are loaded or run. The same block of library code can be shared between several tasks rather than each task containing copies of the routines it uses. The executable is compiled with a library of "stubs" which allow link errors to be detected at compile-time. Then, at run time, either the system loader or the task's entry code must arrange for library calls to be patched with the addresses of the real shared library routines, possibly via a jump table.

Editor

The ADL Editor is a syntax-highlighting editor designed specifically for ADL language source code. As you edit code, the Editor highlights and colorizes different ADL elements, making it easy to follow the flow of your code. You can use the Editor to create new ADL source and function files, or to edit existing ones.

Envox CTI Link for Envox CT ADE

CTI Link integrates Envox CT ADE 9.2 with Envox CT Connect 7.0, allowing CT ADE developers to create call center and IVR solutions with screen pop and intelligent call routing solutions.

Event

Events are actions that you can write code to respond to. For example, there are several events that can be triggered by ADX components, including the ADX Voice Object

IncomingCall and CallerHangup events. Many events have corresponding methods or functions that wait for the specified action to take place.

Function

A function is a sequence of statements that has a name and produces a value. A function can have one or more *arguments*, called *parameters*. Arguments are values that are copied into the function. Inside the function, arguments are referred to by names that behave much like variables except that they may not be assigned values.

Grammars

A grammar is a set of rules that accompany a language and determine which prompts to play during application runtime. The grammar section of the language profile is labeled with the following start and end tags:

<grammar>

</grammar>

The Resource Manager begins processing a phrase element at a specified label in the grammar, comparing the phrase element's Value to a series of patterns. When it finds a matching pattern, The Resource Manager uses the rules that correspond to that pattern to determine which IPF prompts to play. Each grammar entry follows this format:

[Label], Pattern, Rule

Hardware key

A hardware key, sometimes called a dongle, must be present for ADL to run with full functionality. This key allows ADL Studio, the ADL Compiler, and ADL to run on multiple ports of any Resource Manager Resource type.

Indexed Prompt File (IPF)

The Indexed Prompt File (IPF) contains VOX files, also known as prompts, which are recorded words or phrases. Compiled with an .ipf extension, each IPF begins with a header that includes an indexed listing all of the included prompts. The IPF (.TZP file) is a profile include file that lets you specify which Indexed Prompt Files are opened when the Resource Manager starts (also see Profile Include Files). These IPFs can be played directly, and they can also be used by the Resource Manager International Phrases.

IPF (see Indexed Prompt File)

Interactive Voice Response (IVR)

IVR is a telecommunications system, prevalent with PBX and voice mail systems, that uses a prerecorded database of voice messages to present options to a caller, typically over telephone lines. User input is retrieved via DTMF tone key presses (see DTMF). When used in conjunction with voice mail, these systems typically allow callers to store, retrieve, and

route messages, as well as interact with an underlying database server which may allow for automated transactions and data processing.

Method

Methods are used like functions to implement actions in an ADX application. For example, you can use the Send method of Microsoft MAPIMessages control to send an e-mail message. Most of the actions that you want to perform can be enabled using methods in ADX. For example, the SetLogLevel and GetLogLevel methods set and check the level of detail to use when logging fax operations.

Mu-law Encoding (also see A-law Encoding)

Mu-law encoding refers to the assignment of a loudness value to an audio signal using a mu-law translation table. Digital telephony equipment usually stores and transmits audio signal in 8-bit samples using PCM (Pulse Code Modulation) encoding. The value of the 8-bit sample corresponds to the amplitude (volume or loudness) of the audio signal at the time the sample was taken. A translation table is used to determine the amplitude given the sample value. The two tables in common use are the A-law table, which is widely used in Europe and Asia on E-1 trunks, and the Mu-law (m-law) table, which is used in the US, Canada, and several other countries.

Multithreaded application

In ADX, a *thread* is an execution path through an executable program. Most programs have only a single thread. 32-bit Windows allows programs to start additional threads. These *background* threads (as opposed to the *primary* or main thread, which usually controls the user interface) can be used for tasks such as printing or recalculating. Running on a separate thread allows you to continue working in the program. Without a background thread, you may find yourself waiting and looking at an hourglass cursor for the duration of the operation.

Operator Intercept

When an invalid number is dialed or an error condition occurs on the network, an operator intercept may occur. In the US, SIT tones are heard and then followed by a recorded message explaining the problem.

Out-Of-Band Signaling

Out-of-band signaling refers to signaling that is separated from the channel carrying the information—the voice, data, video, etc. Typically the separation is accomplished by a filter. The signaling includes dialing and other supervisory signals.

PABX (Private Automatic Branch Exchange)

Originally, PBX referred to a switch inside a private business (as opposed to one that serves the public). Such a PBX was typically a manual device, requiring operator assistance to complete a call. Then the PBX went modern (that is, automatic) and no operator was needed

any longer to complete outgoing calls. Now you can simply dial 9 to make an outgoing call. Now all PABXs are modern and a PABX is commonly referred to as a PBX.

PBX (also see PABX)

PBX refers to a private (meaning that you own it—not the phone company) Branch (meaning it is a small phone company central office), exchange (a central office was originally called a public exchange, or simply an exchange). In other words, a PBX is a small version of the phone company's larger central switching office.

Personality

In Text-To-Speech, a *personality* describes a single voice that can be used to speak text. Personalities on your system can use different speech engines, speak different languages, or simply speak with different accents.

Phraser.exe

Phraser.exe is a utility that lets you test specific words and phrases from your Resource Manager language. This is especially useful when you are creating a new language and you want to test cases where the outcome may be hard to predict, such as the phrase 'twelve midnight.'

POTS (Plain Old Telephone Service)

POTS is the basic service supplying standard single line telephones, telephone lines, and access to the public switched network.

Predictive Dialing

Predictive dialing is an automated method of using a computer to make numerous outbound calls and then passing answered calls to a person.

Profile ID (also see Profile include file)

Entries in the Profile are called Profile IDs and each has a name and a value. The name of an entry is similar to a path name in a file system. All names begin at the root, which is designated by a back-slash character (\).

The contents of the Profile can be viewed by using the ProfileConfig.exe -L command, which generates a text file that lets you view the contents of the Resource Manager Profile. Each entry in the generated file represents a single Profile ID.

Profile include file (also see Profile ID)

Also called TZP files, profile include files are text files containing information that gets added to the Resource Manager Profile the next time the Resource Manager is loaded. For example, you can use a profile include file to add a new language to the Resource Manager. You can create new profile include files or edit existing ones using any text file editor such as Notepad or the ADL Editor.

Property

A property is a data value similar to a variable and is generally used for attributes of an object that tend to remain fixed. Typical object properties might be BackColor and Font for an object that is displaying a button on the screen. Properties are specified using the name of the object and the name of the property separated by a period.

Resource

The Resource Manager is built around the concept of a Resource. A Resource Manager Resource produces and/or consumes a single stream of audio. There are six groupings of Resource Manager Resources: Trunk Interface Resources, Media Resources (player / recorders), Fax Resources (sender / receivers), Text-To-Speech Resources, Voice Recognition Resources, and Conferencing Resources.

Resource index number

When ADL and the Resource Manager are started, each Resource is assigned a Resource index number, from 0 to the total number of that type of Resource on your system minus one.

Resource Manager Scanner

The Resource Manager Scanner is a utility that runs on your system and extracts information about any installed telephony boards as well as their configuration details. The Scanner then builds a database called the Resource Manager Profile and places the extracted information into it. The Resource Manager Scanner must be re-run each time you change your system's hardware or driver configuration.

RegID

RegIDs are values used internally by the Resource Manager. Some RegIDs describe system details and are stored in the Resource Manager Profile. These RegIDs are called Profile IDs (see Profile ID). The Resource Manager uses other RegIDs internally to access technology-level information. These RegIDs do not appear in the Resource Manager Profile, but can be used with the Get/Set functions to retrieve information about the system or to issue a Technology-specific command at runtime.

RLL (see Runtime Link Library)

Routine

A Routine is a series of source code statements that can be invoked by a single name. In Visual Basic, a routine is actually a subroutine. In C++, a routine is known as a function. Other languages also have different names for the same concept.

Resource Scanner (see Resource Manager Scanner)

Runtime Link Library (RLL) (also see Dynamic Link Library)

Runtime Link Library (RLL) add-ins are extensions that expand ADL functionality. ADL comes with a number of RLLs implemented as Dynamic Link Libraries (DLLs) that let you

load executable code modules on demand to be linked at runtime. RLL functions can be called from an ADL application like other ADL functions. For example, the Network Hub RLL lets you send and receive messages to ADL tasks running on remote PCs that are connected via an IP network.

SAPI

Microsoft Speech API (SAPI) is a platform for adding speech recognition and Text-To-Speech capabilities to computer telephony applications.

Seize

Seize is a computer telephony term that refers to the process of getting access to a phone line prior to making a call. On a home phone, the action of taking the phone off-hook by lifting the handset is known as a seize.

Simulated Phone

Simulated Phone works together with the Resource Manager and your PC's speaker and microphone to simulate a telephony board and phone line so that you can develop and test ADL or ADX applications without requiring a telephony board. This can be helpful in situations where you don't have access to a telephony board.

Speech-To-Text (see Voice Recognition)

Speech-To-Text is another name for Voice Recognition.

Technology (Resource Manager Technology)

In the Resource Manager, a Technology is a specific hardware and API combination that creates a given Resource type. The Resource Manager was designed for API transparency so the same set of functions and methods work under all supported telephony APIs and all supported trunk types—but some features are available only under certain Technologies. For example, GlobalCall systems transmit billing rate information, but other trunk interface APIs (LSI, MSI, etc.) don't use protocol-defined billing rates. Consequently, an ADX Voice Object "GetTrunkBilling" method, which couldn't be ported from the R4GcTrunk Technology to other Technologies, would not be API transparent.

Technology Type

Technology Type/Index pairs (TTIPs) are used in the Resource Manager Profile to set up both listen restrictions and listen additions for Free Listen Resources. A TTIP specifies the Technology type and Technology index number of the Resource that the free listen Resource can listen to.

Text-To-Speech (TTS)

TTS is the creation of audible speech from computer readable text. Text-To-Speech (TTS) technology enables a VRU (Voice Response Unit) to convert textual data into speech output so that a voice-processing application can convey information to the caller that is not

pre-recorded. Also called speech synthesis, TTS is particularly useful when you need to be able to communicate a wide array of unpredictable information to callers.

Touch Tone (also see Dual Tone Multi-Frequency)

Touch tone is a former trademark once owned by AT&T to describe a tone used to dial numbers. Now touch tone is used to describe any phone that has “push-button” numbers.

TZP file (see Profile include file)

Vocabulary

In CT ADE, a vocabulary is a set of pre-recorded audio files used to synthesize phrases such as, *“You have five new messages.”* The vocabulary for this message is probably made up of three fragments: You have / five / new messages.

Voice Board

(Also called a voice card or a speech card) A voice board is a computer add-in card that performs voice processing functions. All voice boards have several important characteristics such as a computer bus connection, a telephone line interface, and typically a voice bus connection. They also usually support going on and off-hook, notification of call termination, sending flash hook, and digit dialing. All voice boards support at least one operating system.

Voice Messaging

Voice Messaging refers to a number of actions that include the recording, storing, playing back, and distribution of phone messages.

Voice Recognition (VR)

Also called Speech-To-Text, Voice Recognition is the ability of a machine to understand human speech. When VR is applied to telephony environments, the limited bandwidth (range of frequencies transmitted by a telephone connection) along with factors such as background noise and the poor quality of most telephone microphones limit the ability of current technology to recognize spoken words. Typical systems are able to recognize standard vocabularies of sixteen or so words, including yes, no, stop, and the names of digits.

Voice Response Unit (VRU)

(Also called Interactive Voice Response Unit or IVR) A Voice Response Unit can be thought of as a voice computer. While a computer has a keyboard for entering information, a VRU uses the touchtones from a remote telephone. While a computer uses a screen to display results, a VRU uses a digitized synthesized voice to “read” the screen to the distant caller. An IVR can do anything that a computer can, from looking up train timetables to moving calls around an automatic call distributor (ACD). The only limitation of an IVR is that you can’t present as many alternatives on a phone at one time as you can on a screen because callers can remember only small chunks of information.

WaveTest

WaveTest is a command-line utility that you can use to scan your system for installed wave device. It also enables you to find out what each wave device does. You can use WaveTest with both ADL and ADX.

Word Spotting

The term Word Spotting refers to the ability of an automated voice recognition device to pick out certain selected words from “background” conversation. For example, the recognizer might pick out “no” from “No thanks,” or “five” from “Give me five please.”

Index

A

ADL
- configuring the log file 6
- overview 4
- sample applications 6
- viewing the log file 2

ADL Language
- overview 1

ADL script
- arithmetic operators 5
- arithmetic using decimal points 5
- audio processing 8
- basic telephony functions 8
- comparison operators 7
- constants 4
- creating a touch tone menu 11
- detecting a disconnect 10
- do...until loop 13
- for loop 12
- goto statements 10
- include statements 11
- jump statements 11
- logical operators 6
- logical values 3
- making an outgoing call 9
- numerical values 3
- operators 4
- source code basics 1
- switch...case statements 9
- telephone signaling 8

terminating a call 10
tutorial 1
variables 3
while loop 13
writing complex expressions 4
writing simple expressions 2
ADX
controlling log output 13
events 3
methods 3
overview 5, 1
properties 3
sample applications 6
viewing the log file 11
ADX ASR Object 2
ADX Conference Object 2
ADX Fax Object 2
ADX TTS Object 2
ADX Voice Object 2
ANI 3
arithmetic operators (ADL script) 5
audio processing (ADL script) 8
Audiotst 7

B

built-in functions (ADL script) 14

C

concatenation (ADL script) 8
constants (ADL script) 4
CT ADE
overview 1
CTADE.ini file 13

D

Debugger
- breakpoints 5
- toggle breakpoint 5
- tutorial 1

DNIS 3
do...until loop (ADL script) 13

E

evaluation mode
- running in 3

events (ADX) 3

F

Fixed Point Math RLL (ADL script) 5
for loop (ADL script) 12
functions
- built-in (ADL script) 14
- user-defined (ADL script) 14

functions (ADL script) 14

G

goto statements (ADL script) 10

H

hardware key 8
hardware requirements for CT ADE 11

I

include statements (ADL script) 11
IPFs.tzp file 10

J

jump statements (ADL script) 11

L

Language files
- IPF 15
- Language Profile 15
- TZP files 15

Language.TST file 16
Language.TXT file 16
Languages 14
Languages.tzp file 15
logical values (ADL script) 3

M

Media Toolbox 7, 3
methods (ADX) 3
money values (ADL script) 6

N

numerical values (ADL script) 3

O

online help documentation list 9
operators (ADL script) 4

P

Phrase Element Parameters
- Gender parameter 17
- Language parameter 18
- Type parameter 17
- Value parameter 16

Profile Include files 10
properties (ADX) 3

R

RegIDs 10
Resource Manager
    Language files 14
    Languages 14
    overview 1
    Profile 8
    Profile ID 9
    RegIDs 10
    Resources 3
    Scanner 12
Resource Manager Scanner
    running 12
Resource states 6
    state diagram 7
Resources 3
    Conferencing 6
    Fax 5
    Media 5
    Text-To-Speech 6
    Trunk 5
    Voice Recognition 5
return statement (ADL script) 17
RLLs (ADL) 14

S

Settings file (ADL script) 18
Simulated Phone 6
    ANI 3
    caller name 3

- DNIS 3
- incoming calls 2
- outgoing calls 3
- overview 1
- phone line properties 3

software requirements for CT ADE 11
string concatenation (ADL script) 8
switch...case statements (ADL script) 9

T

technical support 1
telephone signaling (ADL script) 8
troubleshooting
- ADL 2
- ADX 10

TZP files 10

U

user-defined functions (ADL script) 14

W

WaveTest 7
while loop (ADL script) 13