AT&T 585-310-226
Issue 1
Comcode 107287021
March 1995

INTUITY CONVERSANT

Voice Information System
Version 5.0
IRAPI Programming Guide

Notice

While reasonable efforts were made to ensure that the information in this document was complete and accurate at the time of printing, AT&T can assume no responsibility for any errors. Changes and corrections to the information contained in this document may be incorporated into future reissues.

Your Responsibility for Your System's Security

You are responsible for the security of your system. AT&T does not warrant that this product is immune from or will prevent unauthorized use of common-carrier telecommunication services or facilities accessed through or connected to it. AT&T will not be responsible for any charges that result from such unauthorized use. Product administration to prevent unauthorized use is your responsibility and your system administrator should read all documents provided with this product to fully understand the features available that may reduce your risk of incurring charges.

Federal Communications Commission Statement

Part 15: Class A Statement. This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio-frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user will be required to correct the interference at his or her own expense.

Part 68: Network Registration Number. This equipment is registered with the FCC in accordance with Part 68 of the FCC Rules. It is identified by FCC registration number

AS593M-14695-MA-E.

Part 68: Answer-Supervision Signaling. Allowing this equipment to be operated in a manner that does not provide proper answer-supervision signaling is in violation of Part 68 rules. This equipment returns answer-supervision signals to the public switched network when:

- Answered by the called station
- Answered by the attendant
- Routed to a recorded announcement that can be administered by the CPE user

This equipment returns answer-supervision signals on all DID calls forwarded back to the public switched telephone network. Permissible exceptions are:

- A call is unanswered
- A busy tone is received
- A reorder tone is received

Trademarks

4ESS is a registered trademark of AT&T.
5ESS is a registered trademark of AT&T.
AUDIX is a registered trademark of AT&T.
CLEO is a trademark of CLEO Communications.
CONVERSANT is a registered trademark of AT&T.
DEFINITY is a registered trademark of AT&T in the U. S. and throughout the world.
FlexWord is a trademark of AT&T.
Hayes is a trademark of Hayes Microcomputer Products, Inc.
IBM is a registered trademark of International Business Machines.
INTUITY is a registered trademark of AT&T.
LINKix is a trademark of CLEO Communications.
OBJECT*SQL is a trademark of Oracle Corporation.
ORACLE is a trademark of Oracle Corporation.
ORACLE*Terminal is a trademark of Oracle Corporation.
PRO*C is a trademark of Oracle Corporation.
Smartmodem is a trademark of Hayes Microcomputer Products, Inc.
SQL*Forms is a trademark of Oracle Corporation.
SQL*Menu is a trademark of Oracle Corporation.
SQL*Net is a trademark of Oracle Corporation.
SQL*Plus is a trademark of Oracle Corporation.
SQL*Report Writer is a trademark of Oracle Corporation.
UnixWare is a registered trademark of Novell, Inc.
Voice Power is a trademark of AT&T.

Ordering Information

The ordering number for this document is 585-310-226. To order this document, call the GBCS Publications Fulfillment Center at 1-800-457-1235 (International callers use 1-317-361-5353). For more information about AT&T documents, refer to the *Global Business Communications Systems Publications Catalog* (555-000-010).

You can be placed on a Standing Order list for this and other GBCS documents you may need. Standing Order will enable you to automatically receive updated versions of individual documents or document sets, billed to account information that you provide. For more information on Standing Orders, or to be put on a list to receive future issues of this document, please contact the AT&T GBCS Publications Fulfillment Center.

Comments

To comment on this document, return the comment card at the front of the document.

Acknowledgment

This document was prepared by the GBCS Product Documentation Development group, AT&T Bell Laboratories, Denver, CO 80234-2703.

Contents

Contents

Contents

Contents

Contents

Contents

About This Book

Purpose

This book is a reference for people who develop applications for the AT&T Intuity™ CONVERSANT® Voice Information System (VIS) using the Intuity Response Application Programming Interface (IRAPI). This book provides the necessary background and reference information to use the IRAPI to its potential.

Intended Audiences

The primary audience for this book is experienced application developers and programmers responsible for creating C applications for the VIS environment via IRAPI capabilities. These application developers can be placed into the following categories:

- Sophisticated end customer application developers — This group is responsible for creating and maintaining complex applications for the VIS.
- Application distributors — This group implements and distributes applications for end customers. This includes both internal and external customers.

How to Use This Book

This book is organized to first provide general information about the IRAPI library (Chapter 1 and Chapter 2). To immediately begin developing an IRAPI application, go to Chapter 3, “IRAPI Run-Time Services.” Use Chapter 3 in conjunction with Appendix A, “Reference Material.” Appendix A provides complete information (in the form of manual pages) on all of the functions and data formats supported for the IRAPI. Appendix B, “Sample Applications,” provides several sample IRAPI applications that can be used during the development of your IRAPI application. Chapter 4, “Application Management,” provides additional information needed to compile, install, and debug an IRAPI application. Chapter 5, “Performance and System Tuning for IRAPI Applications,” provides information that can be used to tune an IRAPI application.

How This Book Is Organized

This book is organized into the following chapters:

- Chapter 1 — “Introduction to the IRAPI”

  This chapter provides a brief overview of the IRAPI and some of the basic concepts associated with it. This information includes a description of how the IRAPI is organized in relation to the rest of the voice system, basic terminology associated with the IRAPI, and a system-level architectural description.

- Chapter 2 — “Application Control”

  This chapter describes the Application Dispatch (AD) process that controls applications. This includes starting applications via the AD process and changing the content of the AD tables via the AD-Application Programming Interface (API).

- Chapter 3 — “IRAPI Run-Time Services”

  This chapter describes the use of the run-time services available to an application via the IRAPI. This chapter provides the basic structure of IRAPI applications using the chantest application as an example. This chapter groups each of the run-time services by functional area.

- Chapter 4 — “Application Management”

  This chapter describes the procedure for compiling and linking, installing, and debugging an IRAPI application.

- Chapter 5 — “Performance and System Tuning for IRAPI Applications”

  This chapter contains performance information related to the IRAPI applications and a list of the Resource Manager’s (RM) tunable parameters.

- Appendix A — “Reference Material”

  This appendix contains manual pages for the various library functions, data formats, etc., that make up the IRAPI library.

- Appendix B — “Sample Applications”

  This appendix contains several sample IRAPI applications.

The book also includes a list of abbreviations, a glossary, and an index.

Conventions Used in This Book

The following typographic conventions are used in this book:

- A parenthetical number immediately following the name of a command, function, or data format specifies its type. For example:
  - — (3IRAPI) is an IRAPI function
  - — (4IRAPI) is data format
  - — (1) or (1VIS) is a command
- Symbols use the following notations:

Table 1. Symbol Naming Conventions

Symbol	Description	Manual Page Reference (Appendix A)
IRA_	Algorithm type	*IrALGORITHMS (4IRAPI)*
IRD_	General symbol	*IrDEFINES(4IRAPI)*
IRER_	Error found in the global variable in *irError*	*IrERRORS(4IRAPI)*
IRE_	Event	*IrEVENTS(4IRAPI)*
IREM_	Event modifier	*IrEVENTS(4IRAPI)*
IRF_	Flag (usually a bit mask)	*IrEvent(3IRAPI)*
IRP_	Parameter	*IrPARAMETERS(4IRAPI)*
IRS_	Library state	*IrSTATES(4IRAPI)*

- User input
  - — The word *enter* means to type a value and press ENTER. For example, an instruction to type **y** and press ENTER is shown as

    Enter **y** to continue.

  - — The word *type* means to press the key or sequence of keys specified. For example, an instruction to type **y** is shown as

    Type **y** to continue.

    Do *not* press ENTER after you type the value specified.

  - — The word *select* is used to mean the following: move to the desired menu item using the arrow keys and press ENTER. For example, an instruction to select an item from a menu and press ENTER is shown as

    Select Configuration Management from the Voice System Administration menu.

  - — Information that you enter or type from your terminal keyboard is shown in **bold** type; for example

    Enter **root** at the Console Login prompt.
- Command and file names and their parameters are shown in bold type. Variable parameters are shown in *bold italic* type when they are part of a user input and in *regular italic* type when they are not. All are illustrated in the following example:

  Use the **print** command to print your report. The command syntax is **print *reportname***, where *reportname* is the name of the report to be printed.
- Functions and data formats are shown in *italic* type, for example, *irWait(3IRAPI)*.
- Ellipses (...) after an argument mean that more than one argument may be used on a single line.
- Command options and arguments that are optional are enclosed in brackets — [].
- A list of options for a single argument are separated by vertical bars (for example, a | b | c | d).
- Mandatory argument or identifiers are displayed within parentheses — ().

Related Resources

The following books are to be used in conjunction with this book:

- *Intuity CONVERSANT VIS V5.0 Application Development*, 585-310-227 — This book describes application development using the transaction state machine (TSM) script language. This book is referred to when issues related to TSM script language are discussed.
- *Intuity CONVERSANT VIS V5.0 Script Builder*, 585-310-727 — This book describes application development using the VIS Script Builder. This book is referred to when issues related to Script Builder are discussed.

Refer to the *Intuity CONVERSANT VIS V5.0 Documentation Guide*, 585-350-002, for a complete list of Intuity CONVERSANT documentation.

Technical Updates

Every effort was made to ensure that the information contained in these books is technically accurate, and will guide readers in the normal operation of the system. There are instances however, when the Intuity CONVERSANT VIS V5.0 product may behave differently than is documented in the core library, or when hardware changes are made after these books have been published.

To help with this, an online bulletin board is available to all Intuity CONVERSANT VIS V5.0 customers that provides supplemental information about this product in an electronic format. These updates include hints, tips, and exception conditions about all aspects of the Intuity CONVERSANT VIS V5.0 product that were discovered after the core library was published.

This service is called Access, and is available 24 hours-a-day, seven days-a-week to anyone who subscribes to it. To begin receiving electronic Intuity CONVERSANT VIS V5.0 Access articles, call 1-800-242-6005, and ask for department 186.

Trademarks and Service Marks

The following trademarked products are mentioned in the Intuity CONVERSANT VIS library:

- AUDIX, CONVERSANT, DEFINITY, 5ESS, and 4ESS are registered trademarks of AT&T.
- Voice Power, Intuity, and FlexWord are trademarks of AT&T.
- UnixWare is a registered trademark of Novell, Inc.
- ORACLE, ORACLE*Terminal, OBJECT*SQL, SQL*FORMS, SQL*Menu, SQL*Net, SQL*Plus, PRO*C, and SQL*Report Writer are trademarks of the Oracle Corporation.
- IBM is a registered trademark of International Business Machines.
- CLEO and LINKix are trademarks of CLEO Communications.
- Hayes and Smartmodem are trademarks of Hayes Microcomputer Products, Inc.

How to Make Comments About This Book

A reader comment card is included following the title page of this book. While we have tried to make this book fit your needs, we are interested in your suggestions for improving it and urge you to complete and return a reader comment card.

If the reader comment card has been removed from this book, please send your comments to:

AT&T
Product Documentation Development
Room 22-2C11
11900 North Pecos Street
Denver, Colorado 80234

Please include the name and document number of this book.

Introduction to the IRAPI

1

What's in This Chapter

This chapter provides a brief overview of the Intuity Response Application Programming Interface (IRAPI) and some of the basic concepts associated it. This information includes a description of how the IRAPI is organized in relation to the rest of the platform, terminology associated with the IRAPI, and a system-level architectural description.

Library Overview

The interface provided by the IRAPI offers a standard development interface for voice-telephony applications. The IRAPI provides a high-level, C-language interface to accomplish both voice-processing and telephony functions. IRAPI's capabilities include:

- Voice recording, storage, and play
- Telephone touch-tone sending and receiving
- Telephony call progress
- Speech recognition
- Text-to-Speech (TTS) processing
- Resource management
- Time-division multiplexing (TDM) bus management (bridging and monitoring channels)

The IRAPI includes a general mechanism for starting applications in response to network events.

Library Parameters

The IRAPI is designed to allow application developers to write applications easily, while at the same time provide a rich set of options. In any API, these two goals can be in conflict. In order to avoid burdening commonly used functions with many parameters, library parameters are used to control seldom-needed, minor variations of a function's behavior. These parameters can be queried and set before the function is invoked. All parameters have defaults suitable for most applications. Parameters may be channel-specific or system-wide.

Application Structure and Control

The IRAPI supports voice applications that can serve multiple telephone channels simultaneously. Multi-channel applications typically manage each channel of the application independently. This usually involves maintaining state information for each channel.

NOTE:
Single-channel applications do not necessarily extend to a multi-channel applications easily, but multi-channel applications are often almost as easy to write as single-channel applications.

Most IRAPI functions that request voice and telephony services are asynchronous (that is, non-blocking) functions. These functions return immediately after initiating a service request rather than blocking until the service is complete. When a function returns, control is returned to the application so that it can perform other duties while the requested service is being carried out. These other duties may include servicing a separate telephone channel, accessing a host, or querying a database.

Control is passed between the IRAPI and the application. Applications pass control to the IRAPI so that the library can service requests from hardware devices in real time. The IRAPI passes control to the application by generating events. An *event* is the notification that the IRAPI gives to an application when some condition occurs. In the standard case, applications block in the IRAPI until an event is generated. Events are generated by many conditions. Most importantly, events are generated when asynchronously requested services are completed. Examples of service-completions include completing voice playing or recording, sending touch-tone digits, and the timeout of the IRAPI clock.

The IRAPI allows an application to associate an event with the specific service request (that is, specific IRAPI library calls) using a ‘‘tag’’ with the specific service requests from the IRAPI. Applications pass a tag to the IRAPI when voice or telephony services are requested. The tag is included with the event information returned to the application.

The IRAPI allows applications to control which events are reported as well as which conditions cause interrupts. An *interrupt* is the termination of voice/telephony functions when some condition occurs. Most IRAPI interrupts are controlled by the application, but there are some conditions that cause voice functions to terminate automatically (such as reaching end of a file during a play). Internally, interrupt processing for certain events (notably touch-tone arrivals) is handled as a special case to minimize response time to the event. Note that IRAPI interrupts are not the same as UNIX system events.

Resource Allocation

The IRAPI provides applications with access to abstract capabilities and does not require the application to directly manage the hardware resources required to provide those capabilities.

The IRAPI attempts to make resources available when the application calls functions that implicitly require resources (for example, play or record). When a channel frees the resource, the IRAPI makes it available to other applications running on the platform. In addition, the IRAPI supports reserving, freeing, and querying of resources. Where required, an application may pre-allocate resources to guarantee that those resources are available when needed. when applications need to be isolate applications from each other to avoid resource contention, an application can be restricted to use a only subset of available resources.

The IRAPI provides consistent resource allocation failure modes for both explicit and implicit requests for dynamic resources. If a required resource is not immediately available, applications can fail the request immediately, arrange to wait for the resource forever, or wait for the resource for some fixed period of time. In the last case, if the resource is not available within the time period, the application is notified of the failure with an event.

Library states are used by the IRAPI library to maintain information about channel activities and to prevent applications from attempting illegal operation sequences. In most cases, applications respond to the events as they occur and seldom need to know the library state.

Voice Input and Output

Voice objects can be played or recorded to/from memory buffers, UNIX files[1], or voice file descriptors.[2] Speech input and output is under the complete control of the application. It can be stopped by the application explicitly or implicitly by an interrupt. During play and coding, the IRAPI can notify the application of the progress of the action via events. Voice file descriptors can be opened, closed, positioned, and converted to UNIX file descriptors. Applications can query speech files to determine the coding algorithm and convert speech files from one algorithm to another. Internal components of the IRAPI are responsible for managing the real-time interface between the filesystem and resource cards [for example, a signal processor (SP) circuit card]. In most instances, the platform reduces the chance that gaps in speech requests may occur by queuing up speech files for continuous play. The IRAPI includes capabilities to speak numbers and characters with correct inflections. Applications have a similar interface and level of control over TTS activities.

1 Unlike previous Intuity CONVERSANT VIS generics, speech is stored in standard UNIX filesystems instead of the highly specialized voice file system. In order to support existing applications and packages that rely on storing phrases in the voice filesystem, the IRAPI supports the old voice filesystem semantics via the UNIX filesystem. This involves mapping talkfile and phrase numbers to UNIX files and *vice versa.*

2 These are very similar to UNIX file descriptors.

Telephony

The IRAPI provides basic telephony for a variety of signaling interfaces. Applications can answer incoming calls, place outbound calls [with several options for call classification analysis (CCA)], query and set per-call information such as automatic number identification (ANI) and dialed number identification service (DNIS), dial dual-tone multi frequency (DTMF) digits, flash, and hang up. The IRAPI handles the specifics of the telephony type for the application. In cases where a telephony action is not supported for a given telephony type assigned to a channel, the library reports that the operation is unsupported.

Input Queue and Speech Recognition

Touch tones are collected in a unified input queue[3] that can be manipulated in a variety of ways. The IRAPI supports a flexible built-in mechanism for editing input digits, delimiting sequences of digits, timing user responses for the first and subsequent touch-tone digits, and alerting the application when certain input criteria are reached.

Applications have complete control over speech recognition. Recognized strings are returned via the input queue and therefore have access to all of the input queue features. In addition, applications can use echo cancellation to improve recognizer accuracy when speech recognition is required during voice play. Applications can control the interruption of speech after receiving input.

TDM Timeslot Management

The IRAPI provides functions for managing the time-division multiplexing (TDM) bus and network interface connections to the bus. Applications running on several channels can bridge their TDM time-slots together in a variety of ways. An application can monitor an arbitrary channel, which allows an application to listen to all input and output on that channel. An IRAPI feature also allows starting, stopping, and controlling the volume of background recording. Applications can allocate timeslots and start activities on them.

3 The same input queue is used for speech recognition and touch tone input.

Channel Ownership

The IRAPI uses a *default owner* to internally manage channel ownership. The default owner is an application that is notified when a channel is freed and there are no other pending requests that this channel will satisfy. Typically, the default owner is the process that is responsible for ''listening'' for new calls and dispatching applications in response to them (see Chapter 2, ''Application Control''). Any process can become the default owner for a channel.

Applications can negotiate to acquire specific channels or a channel from a group of channels. As with resources, applications can choose not to wait, to wait for a fixed period of time, or to wait indefinitely for a channel.

Types of IRAPI Processes

IRAPI applications can be processes that start and initialize themselves before they are actually needed by any caller (called *permanent* processes) or they can be dynamically created only when needed (called *transient* processes). Any number of applications of each type can be configured or be actively running on any system.

The IRAPI includes a family of functions that allow applications of any type to invoke one another. These functions model the UNIX *exec(2)* function and allow one application to replace another from the caller's point of view. This interface is flexible enough to allow IRAPI applications to pass control to transaction state machine (TSM) based applications. When processes invoke one another they can pass information to the invoked process. This facility supports both standard information such as ANI and DNIS as well as user-definable information.

IRAPI Organization

The IRAPI is composed of several elements. The Resource Manager (RM) pseudo-driver manages system resources and call profiles for each active channel. The call profile is a collection of data about a call including the channel number, ANI, DNIS, and the start time of the call. The voice response output process (VROP) manages interactions between the filesystem and the speech cards. The IRAPI library is linked to processes that use the IRAPI. The library uses the UNIX kernel, device drivers for signal processor (SP) and network interface (NI) cards, VROP and RM to manage applications. The library communicates with the maintenance subsystem via the logger. The Application Dispatch (AD) process is responsible for examining new calls for DNIS, ANI and channel information in the call profile and starting the appropriate application based on the call profile.

All processes that use the IRAPI are considered IRAPI applications. TSM is a multi-channel IRAPI application that runs applications that are driven by information in scripts. These scripts may be generated by the Script Builder (SB) process. SB uses the IRAPI to manage its pool of standard speech. Other applications that use the IRAPI might be user processes, custom applications or conventional Intuity CONVERSANT data interface processes (DIPs).

Figure 1-1 shows how the IRAPI is organized in relation to the system software processes. Figure 1-2 shows how the IRAPI fits into the voice system architecture.[4] The IRAPI is linked into all processes that use it. Instances of processes linked to the IRAPI maintain per-process private data for managing the application/platform interface.

Several processes are shown in Figure 1-2 using the IRAPI facility:

- Transient process

  This is an application that starts when invoked only when an application should run on a channel. The application can start in response to an incoming call or in preparation to make an outbound call. The application uses the IRAPI to interface to the voice system. A transient process can handle one or more channels.

- Permanent process

  This is an application that starts once (usually at system startup). The application uses the IRAPI to interface to the voice system. It handles one or more channels of some application.

4 Only the relationships between processes that are relevant to the IRAPI are shown in Figure 1-2. For example, all of the voice system processes invoke the logger.

- SB

  Script Builder uses the IRAPI as part of the speech administration function.

- TSM

  TSM is a multi-channel IRAPI application that runs TAS scripts. Two of TSM's pre-IRAPI functions have been moved to other areas: resource management to the IRAPI and application dispatch to AD.

- AD

  AD receives new call indications and determines what application should be started based on the caller's channel and dialed number, and uses *irExec* to start that application. By default, channel ownership reverts to AD when applications fail abruptly or release channels.

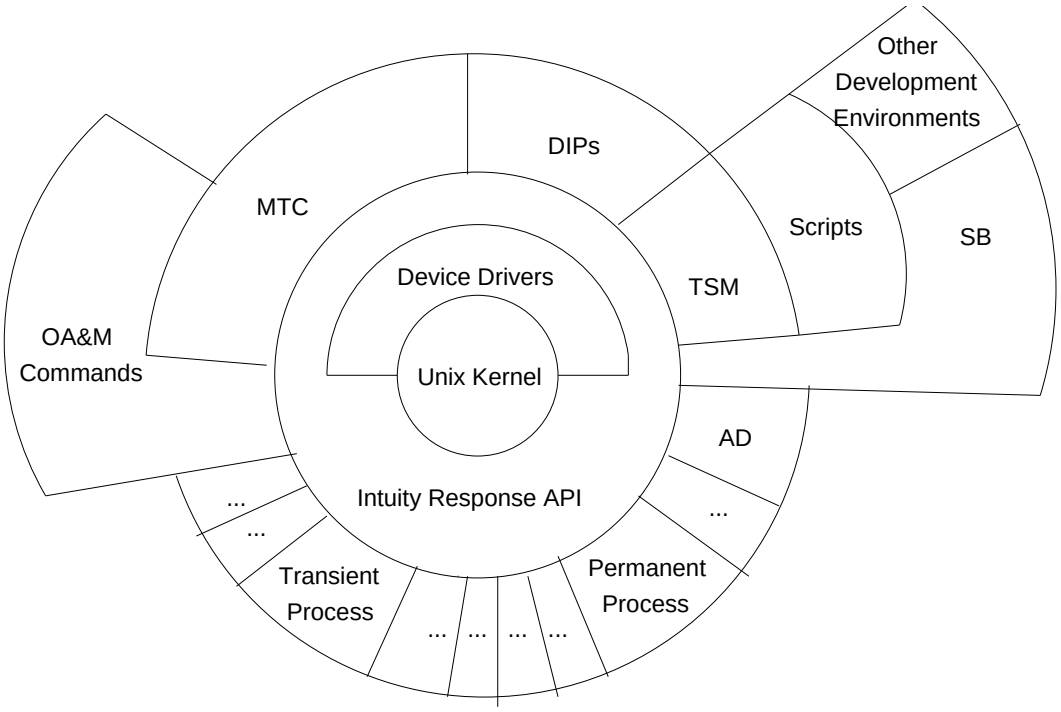

Figure 1-1. IRAPI Organization

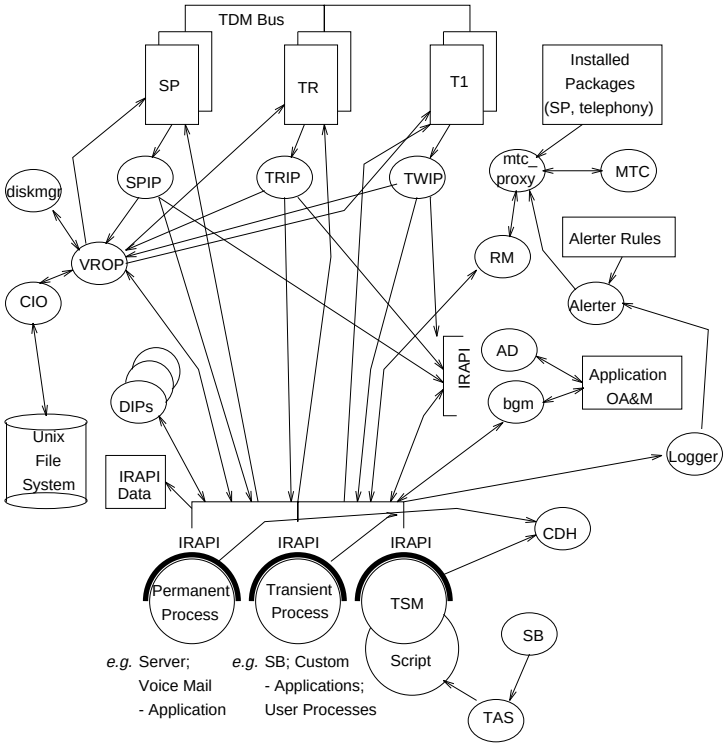

Figure 1-2. IRAPI Architecture Overview

IRAPI Run-Time Architecture

The following describes the basic architecture of the IRAPI:

- The run-time architecture is built in layers to support a more powerful platform for developing applications. From the IRAPI point of view, many elements of the software architecture are sophisticated applications. For example, the TSM and AD processes are standard IRAPI applications. This means they can co-reside (or be replaced completely) by other IRAPI applications.

  TSM is an example of a horizontal-application: it is an application that runs other applications. The applications run by TSM support a particular problem domain. Application developers can create alternative horizontal applications to support solutions for different problem domains.

In contrast, vertical applications directly solve particular problems. These applications can coexist and interwork with other horizontal and vertical applications. For example, an application developer could build a vertical application to provide voice mail to employees or banking services to customers.

- Since the IRAPI is accessible from C, the full capabilities for developing C applications under UNIX are available to the application designer. The IRAPI is designed to be a vehicle to effectively develop co-resident applications.
- The IRAPI library is packaged as a shared object file. This packaging allows upgrades to the library without forcing the applications to re-compile.
- In the IRAPI, application control and resource management is distributed over many processes in the system; the IRAPI coordinates all the applications. The application dispatch function of TSM — determining which application to start based on new call information — is handled by (AD) process. AD starts applications based on scripts as well as IRAPI processes in response to new calls.
- Speech is stored in standard UNIX filesystems instead of the highly specialized voice filesystem. VROP no longer manages the voice filesystem.

  Advanced speech technology packages describe themselves to the system, and that description drives the operation of the system. The maintenance system exploits the capabilities of the logger/alerter to drive automated maintenance of the platform. This makes system maintenance much more automatic and comprehensible.

Distribution of Responsibilities

The following information details the responsibilities of some the system processes based on the introduction of the IRAPI:

Logger

If the IRAPI attempts to control a hardware device and finds an error, it logs the error in the Intuity CONVERSANT logger. The logger accepts the error message and, among other things, forwards it to the Intuity CONVERSANT alerter. Alerter actions are driven by these errors and alerter rules. A standard mechanism for users to introduce application-level dependencies between hardware components are also supported. This allows users to specify card dependencies on a per application basis. A hardware error causes the maintenance process (MTC) to negotiate with the resource manager to take the broken equipment out of service.

Resource Manager

The Resource Manager (RM) keeps the state of all of the voice system resources in private data structures. RM manages all of the static and dynamic resources accessed via the IRAPI. Static resources are channels and resources fixed to a channel (for example, DSP resources on the IVP-4/6 circuit card). Dynamic resources are resources that are not associated with a channel (for example, SP functions for play, code, speech recognition, text to speech and echo cancellation). RM does not *know* anything specific about dynamic resources: the MTC process describes dynamic resources to it as the system starts and as cards move in and out of service. RM simply allocates pieces of these cards from the pools that it maintains to applications that request the resources. The IRAPI functions handle resource allocation. For example, the IRAPI functions that implement recognition with talkoff are responsible for ensuring that the appropriate echo cancellation and recognition resources are acquired and coordinated.

RM is implemented as a pseudo driver (that is, a driver that does not control any hardware) to allow for a simple and reliable strategy for dealing with ungraceful application exits and crashes. When an application starts for the first time, it opens the RM driver. If an application dies unexpectedly, the close entry point for RM driver is invoked automatically by the kernel as it closes each file descriptor associated with the process. This unexpected close signals that an application has terminated abruptly and should have its resources freed and returned to a sane state.

The RM debug/monitor process can be used to query the state of the managed resources.

MTC notifies RM as equipment comes into service (through system startup or OA&M requests). As it does, MTC passes the characteristics of the resources (as described in the installable packfile) to RM. When cards go in and out of service, the resource manager notifies the owners. The description of the capabilities of specific cards or of a specific capability is associated with the installable package for the card or capability. Assignments for capabilities running on multi-purpose cards (for example, speech recognition on an SP circuit card) are maintained by the MTC platform.

Application Dispatch

The AD process listens to new calls that arrive on the network, finds an application to handle the call, and starts the associated application with *irExec*. AD uses call information stored in the AD tables (such as DNIS, ANI, and the port number) to associate applications with calls. The OA&M system can establish, break, and query this association through the IRAPI. AD holds all idle channels and releases them upon receiving a request. For example, the **soft_szr** command causes TSM to request a channel from AD to run a TSM script.

Applications can change the behavior of AD via the AD-API. The AD-API allows users to add entries to the AD table, query the AD table, and remove entries from the AD table. AD can be instructed to dispatch calls for a subset of the channels in the system. It is also possible to substitute an alternative AD that implements some other dispatch logic. For example, an alternative AD might not use fixed tables, but would rather consult an network control point (NCP) over a network to make the routing decision.

When a new call arrives, if AD is the application owner for the channel, it starts the application.

NOTE:
AD need not be the agent for starting applications. Users can develop and use their own processes that handle the AD function, or they can rely on the applications to allocate channels and start themselves up in response to new calls.

Voice Capabilities

Applications use the IRAPI to cause voice activities to happen. Requests to play or code voice data are passed to the VROP process. Plays or codes can use memory buffers, entire UNIX files or UNIX file descriptors as sources and destinations. VROP uses the customer input/output (CIO) processes to read and write information in and out of the UNIX filesystem. In order to actually play or code speech, VROP interacts with the DSP hardware, (for example, SP and Tip/Ring hardware). For language playback (LP), the IRAPI uses the tables to associate recorded phrases with vocabulary items generated by Script Builder (that is, the standard speech pool) to support the process of speaking language-independent alphanumeric strings, numbers, dates, times, and currencies.

Responsibility Restructure Caused by the IRAPI

The following information details the differences in system responsibilities and functions due to the introduction of the IRAPI for Intuity CONVERSANT VIS V5.0:

- TSM

  Previously, TSM managed all of the resources in the system: that role belongs to RM. TSM also started all applications in response to network events, was the owner of all idle channels, and was the arbitrator for channels ownership by applications: that role belongs to AD. In Intuity CONVERSANT VIS V5.0, TSM is written as an IRAPI application and is responsible only for interpreting scripts. Some of the applications that are started by the IRAPI are traditional TSM scripts.

- VROP

  In the previous releases, VROP had several functions:

  — Manage the content of the voice filesystem

  — Use the CIO processes to coordinate the transfer of voice data between the filesystem and voice buffers

  — Coordinate the transfer of speech data between voice buffers and the voice cards

  In Intuity CONVERSANT VIS V5.0 All speech is stored in UNIX filesystems (see information on speech filesystem below) and VROP no longer manages them.

- Voice filesystem

  The voice filesystem was carefully designed to achieve maximum performance. Accessing voice stored in UNIX files is substantially less efficient. On the other hand, accessing voice via voice filesystems is much more difficult because it requires the VROP process to run and because the application interface to VROP is limited. Other changes in the Intuity CONVERSANT architecture — the introduction of the 486 as the sole CPU platform and the use of alternate and larger block size filesystems (like the Veritas 8K file system in UnixWare) — removes some of performance constraints that dictated the use of the voice file system.

  Storing speech in the UNIX filesystem also gives application developers the same semantics for speech as are available for standard UNIX files. Specifically, the concept of file pointers, seek, and append are supported that are not offered with voice filesystem.

  Many existing applications and packages rely on storing phrases in the voice filesystem. In order to allow these applications to continue to function, the system supports voice file system semantics via the UNIX filesystem. This involves mapping talkfile and phrase numbers to UNIX files and vice versa. For example, talkfile 7 phrase 99 might map to the UNIX file:

  /home2/vfs/talkfiles/7/99

  TSM instructions, Script Builder actions and administrative commands that rely on the voice filesystem are changed to support this mapping.

- Voice system card interface

  Requests for action by any voice system card are channeled through a library associated with that card. Because there may be many processes that are interacting with a specific card at any point in time, each card type has its own interface process (IP). The interface process for the SP is SPIP, for the Tip/Ring TRIP and for the T1 (T-won) card, TWIP. The IPs read indications from the cards and forward them to the appropriate processes through IPC messages. Messages associated with a given channel are sent to the *application owner* of the channel. This

relationship between a channel and its application owner is established when the process executes *irInit*. This relationship is tracked by RM and all messages are routed through RM. RM sends the message to the IPC message queue associated with the channel. If there is no owner associated with a channel, RM holds messages for that channel in a deferred queue for a fixed period of time. Usually, the AD process is the default application owner for all of the channels in the system and, therefore, has the responsibility for dispatching applications on all channels.

The IRAPI interfaces directly with each of the cards to provide specific services. Requests for speech recognition, TTS, CCA and primary rate interface (PRI) functions are handled by the IRAPI by invoking functions on SP circuit cards through the SP driver. The IRAPI interfaces directly with the T1 and T/R circuit cards for telephony and touch-tone services through the T1 and T/R drivers. Responses from SPs, T1s and T/Rs are passed to the IRAPI though the IPs.

- Data interface processes (DIPs)

  Applications must use the *irPostEvent* function to call a DIP through the IRAPI. This allows them to use standard IRAPI functions to send messages, wait for responses, and decode the results. The DIPs themselves may still use the conventional **mesgsnd/mesgrcv** interface (refer to Chapter 4, "Data Interface Process" of *Intuity CONVERSANT VIS V5.0 Application Development*, 585-310-227). The **mesgsnd** and **mesgrcv** functions are re-implemented to use *irPostEvent* internally.[5] This interface is supported for compatibility with the TSM-based application model. New DIPs (created to run with Version 5.0 software) should be written exclusively in terms of the IRAPI (that is, using *irPostEvent* and *irWait*).

- OA&M interface

  The IRAPI interfaces with the platform OA&M system to allow the OA&M system to display and track the system monitor state of applications and voice hardware.

- Call data handler (CDH) process

  Only TSM applications can use CDH.

5 All existing DIPs have to be re-compiled for Intuity CONVERSANT V5.0 to pick up the new mesgsnd/mescrcv.

IRAPI with Intuity CONVERSANT VIS Features

The introduction of the IRAPI increases the implementation choices for an application developer without introducing new feature interaction problems.

The IRAPI library supports all the features like TTS, WholeWord speech recognition, FlexWord speech recognition, Adjunct/Switch Application Interface (ASAI), Line-Side T1 (LST1), and PRI that have been introduced in Intuity CONVERSANT generic software in previous releases. As proof of the completeness of the support for other features, note again that TSM has been modified to use the IRAPI library directly rather than lower level libraries to support these features.

Scripts provided by add-on packages like AUDIX Voice Power (AVP) and Fax Attendant can be *irExec*'ed from IRAPI applications much like the same scripts could be exec'ed from other TSM scripts.

Application Organization

Deciding how to implement the application must be decided early in the application design process. You have the following choices:

- Use Script Builder and the TAS script language
- Write a special purpose program for your application
- Write a general-purpose program for your application and cast your application in terms of that program

First, Script Builder and TSM script still solve all of the problems that they used to solve: they make it very easy to build a very large class of applications. The TAS language is an extremely efficient method of designing voice applications. The amount of memory devoted to holding the chantest application is very small: the overhead for a full UNIX process is reasonably large. For larger applications, this advantage disappears.

If you choose to write an IRAPI application, you must choose whether you want the application to control a single channel or multiple channels, and whether you want the application to start only when it is invoked (a *transient* process) or whether you want the application to start when the voice system starts (a *permanent* process) and be ready to quickly handle calls when they come in.

A transient process is created by irExecvp etc. when the exec call is made. This does not use memory for applications that are not actually running, but takes extra time and effort to get the application loaded and running. This time can be significant if the application to be started is large and the platform is otherwise busy. When a transient process is started, the first time that it calls irWait, an IRE_EXEC event is immediately generated. The process should use this event to determine which channel to acquire and on which channel to start the application.

A permanent process is usually started out of inittab at the same time that the voice system is stared. Permanent processes wait for IRE_EXEC messages and respond to them by starting the application running. Since they are started out of inittab, they consume a slot in the process table and some amount of memory (although if the process is not accessed in a long time, pages that are not required are likely to be paged out to the swap device).

All UNIX executables are composed of three basic parts:

- Text — The instructions in the program. This section cannot normally be modified, and therefore each instance of an application that references that particular data section can safely share the same copy of the text.
- Data — The initialized data in the program. This area is dynamic.
- BSS space — The uninitialized data in the program. This area is dynamic and it can grow as a result of **malloc(3)**/**brk(2)** calls.

NOTE:
Since data and BSS areas can be changed, each instance of a process has a unique copy of these areas.

Multiple copies of the same IRAPI-based application use less space whether permanent or transient processes.

The IRAPI is delivered as a UNIX shared object. Shared objects also have text, data, and BSS sections. If more than one application links against the IRAPI all copies of that application use the same IRAPI text space. Unique copies of the data and BSS sections are allocated for the process. The following table shows approximately how much space the Intuity CONVERSANT VIS V5.0 IRAPI library uses.

Table 1-1. Memory Usage

Library	Text	Data	BSS
/usr/lib/libirAPI.so	580000	22000	13000
/usr/lib/libirEXT.so	1000	0	0

Since AD, TSM, and other processes use and link against the IRAPI library, there is no additional memory cost for user applications to use the IRAPI text area.

Keep the following considerations in mind when determining whether to use multi-threaded or single-threaded processes.

- Multi-threaded processes are more memory-efficient than multiple single-threaded processes.
- Increasing the number of processes increases the amount of work the kernel must do to schedule and coordinate them.
- Multi-threaded processes are more processor-efficient than single-threaded processes.

Single-threaded vs. Multi-threaded; Permanent vs. Transient

The following information details the differences between single-threaded and multi-threaded applications and between permanent and transient processes. As noted in the table below, permanent, multi-threaded processes are the most efficient arrangement.

- Permanent processes are loaded and initialized at system startup time; transient processes are loaded and initialized on *irExec*.
- Permanent processes are quicker to load and start processing user inputs Permanent process also use CPU resources more efficiently.
- Single-threaded processes can mix *irWait()*'s throughout their program. This is impossible for multi-threaded applications.
- Transient processes must deal with creating unique names.
- Permanent processes may want to clean out the message queue as they start (as shown below). Transient processes do not want to miss an IRE_EXEC message.

```
while (irCheck () != IRE_NULL)
        ;
```

The following table compares the application characteristics between single-threaded and multi-threaded applications and between permanent and transient processes.

Table 1-2. Application Characteristics

	Single Threaded	Multi-Threaded
Transient	Simplest to design and build	Unusual choice but possible
Permanent	Structure almost identical to single-threaded, transient	Most complex and efficient (for example, TSM)

Application Control

2

What's in This Chapter

This chapter describes the Application Dispatch (AD) process that controls applications. This includes starting applications via the AD process and changing the contents of the AD tables via the AD-Application Programming Interface (API).

Application Dispatch Process

The AD process is a permanent, multi-threaded IRAPI process that starts (or dispatches) an application on a channel when a new call arrives for that channel. By default, the AD process is the default owner of all the channels on the system. The default owner for a channel receives the IRE_DEFOWNER event when another process deinitializes a channel and there is no other process waiting for that channel. In other words, the default owner is the process that accepts ownership of a channel when there is no other process that *wants* the channel. The default owner for a channel is set using the *irDefOwn* function.

When a out-of-service channel is restored to service via the **restore** command, the MTC_PROXY process places the channel on-hook and then calls *irDeinit* for that channel. If there is no process waiting for ownership of that channel, AD receives an IRE_DEFOWNER event for that channel. AD uses *irInit* to initialize the channel and sets the IRP_CHAN_NEGOTIATION parameter to IRD_ALLOW. This allows another process to *irInit* the channel if it desires. AD then calls *irWait* to wait for another IRAPI event.

When the AD process receives an IRE_NEWCALL representing an incoming call for a channel, AD uses the *iraQueryADTables* function to determine what application should be started on this channel.

Application Dispatch API

The AD-API is a subset of functions within the IRAPI that manipulate the AD tables and the associated service registration files.

Application Dispatch Tables

There are two AD tables that are used to determine which application should be dispatched when a call arrives on a particular channel:

- AD Channel table

  This table contains at most two applications per channel: the standard application and the startup application. The standard application is typically the only application used and is displayed for a channel if the user uses the **display channel** or **display card** commands. The startup application is used only when special processing is required when a new call arrives on a channel before the standard application starts up. In this case, an IRAPI application that performs the special processing is assigned to the channel as the startup application and the *regular* application is assigned as the standard application. When a new call arrives, the AD process uses the AD-API function *iraQueryADTables* to determine which application should be started. Since a startup application is assigned, AD *irExec*s the channel to the startup application. The startup application performs the special processing and then either

irExec's the channel back to AD or uses the *iraQueryADTables* function itself (with the IRD_AD_STANDARD argument) to determine which application to *irExec*.

For example, a startup application could be used with the converse vector step for a DEFINITY. Either or both of the standard and startup applications can be null or the special application ''*DNIS_SVC.'' The ''*DNIS_SVC'' application indicates that the AD DNIS/ANI table (described below) should be searched to find the application for this channel. If both the standard and startup applications are null, an error is reported in the error log.

- AD DNIS/ANI table

  This table contains an ordered list of dialed number identification service (DNIS) and automatic number identification (ANI) ranges and associated applications. This table is ordered primarily by DNIS ranges, most to least specific. If two entries have the same DNIS range but different ANI ranges, the entries are ordered by ANI ranges, most to least specific. The order of the AD DNIS/ANI table is important because AD uses the AD-API *iraQueryADTables* function to determine which application to start. If necessary, *iraQueryADTables* searches the AD DNIS/ANI table in order and returns the application for the first entry whose DNIS and ANI ranges contain the DNIS and ANI of the incoming call.

DNIS and ANI ranges

A range is considered most specific if it contains (or matches) only one number. A range is considered least specific if it contains (or matches) any possible number. The notation for DNIS and ANI ranges is *a*:*b*, where *a* and *b* are positive integers and *a* <= *b*. Ranges can completely contain other ranges, but ranges cannot overlap to prevent ambiguity across the overlap ranges. For example, the set of ranges 4000:4000,3500:6000,0000:9999 is valid. But given the previous set of ranges, the range 4500:8000 is invalid because it overlaps the range 3500:6000.

The IRA_STR_RANGE structure contains the DNIS and ANI range information. The *iraSetStrRange(3IRAPI-AD)* function sets the contents of this structure. Some AD-API functions take pointers to this structure as arguments.

Initializing AD Tables

The AD tables are initialized at system startup. Typically, application developers should not initialize the AD tables themselves, but several functions are supplied if they are needed. The *iraInitADTables* function initializes both the AD Channel and DNIS/ANI tables, while the *iraInitADChannel* and *iraInitADDnisani* functions initialize only the AD Channel table or only the AD DNIS/ANI table, respectively. After initialization, all applications in the AD Channel table are set to NULL and the AD DNIS/ANI table is empty. Any previously existing application assignments are lost. The global parameter *IRP_AD_MODE* must be set to *IRD_AD_READWRITE* to initialize the AD tables; otherwise, the functions return an *IRER_PERMISSION* error.

Querying AD Tables

The *iraQueryADTables* and *iraQueryADDnisani* functions can determine which application should be started in response to a new call arrival. The *iraQueryADTables* function uses both the AD Channel table and the AD DNIS/ANI table to determine the application, while the *iraQueryADDnisani* function only uses the AD DNIS/ANI table.

A parameter passed into the *iraQueryADTables* function influences its behavior. If the parameter is IRD_AD_STARTUP, the *iraQueryADTables* function looks at the startup application in the AD Channel table for the particular channel. If the application is not null and not ‘‘*DNIS_SVC,’’ it then returns this application. If the application is ‘‘*DNIS_SVC,’’ then the *iraQueryADTables* function searches the AD DNIS/ANI table to the find first entry that matches the DNIS and ANI for this call. If it finds an entry whose DNIS and ANI ranges match those of the call, it returns the application associated with the entry; otherwise, it returns an error. If the startup application is null, the *iraQueryADTables* function looks at the standard application for the channel.

If the parameter is IRD_AD_STANDARD or if the startup application is null for a particular channel, the *iraQueryADTables* function looks at the standard application for the channel. If the application is not null and not ‘‘*DNIS_SVC,’’ it then returns this application. If the application is ‘‘*DNIS_SVC’’ or null, then the *iraQueryADTables* function searches the AD DNIS/ANI table to find the first entry that matches the DNIS and ANI for this call. If it finds an entry whose DNIS and ANI ranges match those of the call, it returns the application associated with the entry; otherwise, it returns an error.

Reading AD Tables

Applications can read the contents of the AD Channel and DNIS/ANI tables by using the *iraReadADChannel* and *iraReadADDnisani* functions, respectively. The *iraReadADDnisani* function reads the AD DNIS/ANI table in order and return each entry one at a time. Subsequent calls to *iraReadADDnisani* returns the next entry. The *iraRewindADDnisani* function is used to reset (or rewind) the list, so that the next call to *iraReadADDnisani* returns the first entry in the list.

Changing AD Tables

Applications can be assigned to a particular channel or for a particular DNIS/ANI range by using the *iraAddADChannel* and *iraAddADDnisani* functions. The global parameter IRP_AD_MODE must be set to IRD_AD_READWRITE to make changes to the AD tables; otherwise, the functions return an IRER_PERMISSION error.

Service Registration Files

Service registration files are read by the *iraAddADChannel* and *iraAddADDnisani* functions and stored in an *AD_APPL* structure when an application is added to the AD tables. The information contained in the service registration files is used as arguments to the *irExec* function when the application is started. The AD_APPL definition is:

```
typedef struct ad_appl {
    char service[IRD_SERVICE_NAME_LEN];   /* name of the service           */
    char process[IRD_PROCESS_NAME_LEN];   /* name of the process           */
    char register_file[IRD_MAX_FILE_LEN]; /* name of the registration file */
    int type;                             /* type of application           */
    unsigned long attributes;             /* attributes of application     */
    time_t modtime;             /* modification time of registration file  */
} AD_APPL;
```

IRAPI application developers must create the service registration files for their applications and deliver the service registration files along with the application files. Service registration files are created by using the **defService(1)** command (see below).

defService Command

The **defService** command is used by IRAPI application developers to create the service registration file for an IRAPI service. The service registration file is required to assign and/or delete a service to/from a channel or DNIS and/or ANI. See the **assign**, **defService**, and **delete** commands in the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230. For TSM scripts, the output of the **tas** command serves as the service registration file for the script.

If it successful, **defService** creates the service registration file **/vs/trans/**<***service***>*.T*. If the command is called without options, **defService** prompts for all the necessary information. For example, to create the service registration file for the chantest service, enter:

defService -n -p chantest -t P chantest

IRAPI Run-Time Services

3

What's in This Chapter

This chapter discusses the run-time services available through the Intuity Response Application Programming Interface (IRAPI) for the Intuity CONVERSANT Voice Information System (VIS). First, an example IRAPI application (chantest) is presented. This application provides the basic structure or framework applicable to most IRAPI applications due to their event driven nature. The code that implements the chantest application is fragmented throughout the chapter and provides a high level description for each function used. The entire chantest application (chantest.c) is included in Appendix B, ''Sample Applications.''

This chapter also details the IRAPI functions and data structures and how applications can be built using them. Each section is grouped by functional area and example code fragments illustrate how the functions may be used. In some sections, the chantest.c file is expanded to illustrate the additional run-time services available.

This chapter should be used in conjunction with Appendix A, ''Reference Material.''

Application Framework

A well-written IRAPI application must be event-driven rather than procedural. Most IRAPI functions are non-blocking functions. Those that implement voice or telephony functions, for example, initiate the specified action and return immediately. The application must then wait for an event generated by the IRAPI for status or completion information for the initiated action. State information should be maintained for each channel handled by the application. This information determines the step to perform after receiving an IRAPI event.

In general, a well designed IRAPI application has the following components:

- Process initialization
- Application initialization
- Application execution
- Application termination
- Process termination

Following a discussion of each application component is an implementation of each task using the chantest.c code. The chantest application prompts for and collects touch tones. After receiving four touch tones, it plays back the touch tones collected. This behavior is repeated until four zeros (0000) are entered to terminate the application.

Process Initialization

The following tasks should be performed by an IRAPI program when execution starts:

1. Register the process with the system with the *irRegister(3IRAPI)* function. This function must be passed a unique process name. Upon successful completion, a UNIX message queue key is returned.

   For example, the chantest.c application uses the *irRegister(3IRAPI)* function as follows:

   ```
   if ((qid=irRegister("chantest")) < 0) {
       irPError ("Error on irRegister");
       exit (1);
   }
   ```

2. Allocate and initialize per-channel process data. Multi-channel IRAPI applications may use the *irNumChans(3IRAPI)* function to get the number of channels configured in the system and allocate per-channel data structures accordingly.

As chantest.c is a multichannel application, it uses *irNumChans(3IRAPI)* to obtain the number of channels in the system and allocates a CHLDATA data structure for each channel. By allocating one data structure for every channel in the system, it can be assigned to handle any or all channels simultaneously:

```
if (initData(irNumChans(IRD_REAL)) < 0) {
    printf("Initialization failed );
    exit(1);
}
```

The following code shows the chantest's initData() routine:

```
/* Global variables */

struct CHLDATA {
    int State;
    int RetryCount;
    int PlayDone;
    ir_event_t InputDoneEvent;
} *Chl;

int initData(int nchans)
{
    if(nchans <= 0) return(-1);
    if((Chl = (struct CHLDATA *)calloc(nchans,
              sizeof(struct CHLDATA))) == 0)
    {
        return(-1);
    }
    return(0);
}
```

Application Initialization

Before an IRAPI application can start on a specific channel, it must perform the following tasks:

1. Obtain ownership of the channel.

   If the IRAPI application is assigned as a service to one or more channels, it should wait for an IRE_EXEC event from the IRAPI using the *irWCheck(3IRAPI)* function. This event is passed to an application when a new call arrives on a channel to which the application is assigned, or when another application uses *irExec(3IRAPI)* to run the application. When the IRE_EXEC event is received, call the *irInit(3IRAPI)* function to obtain ownership of the channel specified in the IRE_EXEC event data structure [see *IrEVENTS(4IRAPI)* in Appendix A, ''Reference Material''].

If the application is to run on a channel to which it is not assigned, it should first request ownership of the channel with *irInit(3IRAPI)* or *irInitGroup(3IRAPI)*. It should then use *irWCheck(3IRAPI)* to wait for the IRE_CHAN_GRANT event before continuing with the application on that channel. Refer to ‘‘Channel Management’’ later in this chapter.

In each case, *irInit(3IRAPI)* or *irInitGroup(3IRAPI)* is used to obtain a valid channel ID (cid). This cid value is used as an argument to all other channel specific IRAPI functions and is included as part of the event data structure for all events that result from calling these functions.

After process initialization, chantest enters a while loop to wait for IRAPI events with *irWCheck(3IRAPI)*. The chantest *application* is designed to be executed by the Application Dispatch (AD) process when a new call arrives on a channel to which chantest is assigned. Therefore, after process initialization, the chantest *process* does not start running the application until it receives an IRE_EXEC event. All events are handled by the switch statement within the while loop. The following code fragment shows how chantest initializes the application after receiving an IRE_EXEC event:

```
ir_event_t ev;
channel_id cid;
int chan;

while (irWCheck(&ev) != IRR_FAIL) {
    switch (ev.event_id) {
       case IRE_EXEC:
           chan = ev.event_mod1;
           if (irInit (chan, &cid, IRD_IMMEDIATE, 0) != IRR_OK) {
              irPError ("Error on irInit");
              break;
           }
           Chl[chan].State = BUSY;

           if(setEvents(cid) < 0) {
              cleanup("Error on setEvents", cid);
              break;
           }
           if(setTTParams(cid) < 0) {
              cleanup ("Error on setTTParams", cid);
              break;
           }
           .
           .
           .
    }
}
irPError("irWCheck Failed.");
exit(1);
```

After receiving an IRE_EXEC event, chantest gets the channel number from the **event_mod1** modifier in the event data structure and attempts to obtain ownership of the channel and a valid channel identifier (cid) with *irInit(3IRAPI)*.

2. Any application specific data should be allocated and/or initialized after channel ownership is obtained.

3. Set the initial disposition of any IRAPI events used by the application with *irSetEvent()* if the default disposition is not used. See *irEvent(3IRAPI)* and *IrEVENTS(4IRAPI)* in Appendix A, ''Reference Material.'' With *irSetEvent()*, events may be ignored or interrupt telephony or voice activity on the channel when they occur.

   The dispositions of the IRE_INPUT, IRE_INPUT_DONE, IRE_WINK and IRE_DISCONNECT events are set with chantest's setEvents() subroutine. The IRE_INPUT and IRE_INPUT_DONE events are simply set to notify chantest when they occur. These events indicate caller input. The IRE_WINK and IRE_DISCONNECT events indicate call termination. Chantest sets them to notify of their occurrence and to interrupt any voice play activity on the channel when they occur.

```
int setEvents(channel_id cid)
{
    /* Enable events
     */
    if ( irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_WINK, IRF_NOTIFY | IRF_PLAYINTR)
             == IRR_FAIL ||
      irSetEvent(cid, IRE_DISCONNECT, IRF_NOTIFY | IRF_PLAYINTR)
             == IRR_FAIL ) {
       return(-1);
    }
    return(0);
}
```

4. Set any IRAPI parameters to be used by the application with *irSetParam* or *irSetParamStr* [see *irParam(3IRAPI)* in Appendix A, ''Reference Material''].

   Chantest uses its setTTParams() subroutine to set initial values of four IRAPI caller input parameters. The values of these parameters determine when an IRE_INPUT_DONE event is generated:

```
int setTTParams(channel_id cid)
{
    /* Set up for reading INPUT_LEN digits from input queue
     */
    if ( irSetParam(cid,IRP_INPUT_LEN,INPUT_LEN) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_PRETIME, 8000) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_INTERTIME, 5000) == IRR_FAIL ||
      irSetParamStr(cid, IRP_INPUT_DELIM1, "#") == IRR_FAIL ) {
        return(-1);
    }
    return(0);
}
```

The IRP_INPUT_LEN parameter specifies the maximum input length (INPUT_LEN is defined as 4 characters). The pre-digit and inter-digit timeout parameters are set to 8000 and 5000 milliseconds (8 and 5 seconds) respectively. The pound sign (#) is set as an input delimiter for input shorter than IRP_INPUT_LEN.

Application Execution (Event Loop and Switch)

After application initialization, the application is ready to perform whatever IRAPI functions necessary to interact with the caller in the way defined by the application. Usually, the best way to implement this interaction is through an event switch inside a continuous loop on the *irWCheck(3IRAPI)* function. Each case in the switch should handle a unique event, perform the next event generating function, and return to the wait loop [irWCheck()]. Each time an application executes an event generating function, it should not execute another IRAPI function until it receives the event that terminates that function.

For example, an application might perform the following functions after application initialization:

1. Answer the phone with *irAnswer(3IRAPI)* and wait for the IRE_ANSWER_DONE event.

   In the chantest.c application, after the IRE_EXEC event is received and chantest initializes the application, it answers the incoming call with *irAnswer(3IRAPI)*. It then returns to the while loop to wait for the IRE_ANSWER_DONE event. The IRE_ANSWER_DONE event contains the result of the answer attempt.

```
while ( irWCheck(&ev) != IRR_FAIL ) {
    switch(ev.event_id) {
        .
        .
        .
      case IRE_EXEC:
            .
            .
            .
        if (irAnswer(cid) == IRR_FAIL) {
            cleanup ("Error on irAnswer",cid);
        }
        break;

      case IRE_ANSWER_DONE:
        if ( ev.event_mod1 != IREM_COMPLETE ) {
            cleanup("Error in IRE_ANSWER_DONE", cid);
            break;
        }
        startChanTst(cid);
        break;
```

2. Queue up voice files to play with *irFPlay(3IRAPI)*, start play with *irEnd(3IRAPI)*, and wait for the IRE_PLAY_DONE event.

   Chantest uses its startChanTst() subroutine when the IRE_ANSWER_DONE event is received to perform this step of the application. This subroutine plays the introductory announcement, ‘‘Begin testing.’’

   ```
   void startChanTst(channel_id cid)
   {
       int chan = irCid2Chan(cid);

       Chl[chan].PlayDone = IGNORE_WHEN_DONE;
       Chl[chan].RetryCount = 0;

       if (irFPlay(cid, 0, "/speech/begn.tst") < 0) {
           cleanup ("Error on irFPlay", cid);
           return;
       }
       playInstr(cid);
   }
   ```

The startChanTst routine converts the cid to a channel number with *irCid2Chan(3IRAPI)*. It uses the channel number to set application specific data for the channel. The PlayDone flag indicates what is to be done when the next IRE_PLAY_DONE event arrives. The RetryCount indicates the number of times the application has prompted the caller after getting no input. *irFPlay(3IRAPI)* is used to queue up the ‘‘begin testing’’ introductory phrase stored in the UNIX file **/speech/begn.tst** for playing. The playInstr() subroutine then is called to play the instructions that are part of the initial chantest prompt.

```
void playInstr(channel_id cid)
{
    int chan = irCid2Chan(cid);

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

   if(irFlushInput(cid) == IRR_FAIL ||
    irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
    irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
    irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ||
    irFPlay(cid, 0, "/speech/all.rptd") == IRR_FAIL ||
    irFPlay(cid, 0, "/speech/term.4.0") == IRR_FAIL ||
    irEnd(cid, 0, 0) < 0) {
      cleanup ("Error in playInstr", cid);
   }
}
```

Before the prompt is played, the PlayDone flag is set to indicate that the touch-tone input timer should be started when the next IRE_PLAY_DONE event is received. The *irFlushInput(3IRAPI)* function clears any caller input that has been given before the prompt. This synchronizes caller input with prompts. The *irSetEvent(3IRAPI)* function is used to set the IRE_INPUT event to interrupt speech play when the event is generated. This enables the ''talkoff'' feature, allowing the caller to interrupt a prompt with touch-tone input. Three separate phrases are queued up with *irFPlay(3IRAPI)* that make up instructions to the caller: ''Enter 4 touch tone digits,'' ''All digits except star and pound will be repeated,'' ''Terminate your input with a pound sign.'' Speech play is started with the call to *irEnd(3IRAPI)* and the subroutine returns back to the main while loop to wait for the next event.

3. Start the touch-tone input timer with *irStartTTTimer(3IRAPI)*, collect input with *irGetInput(3IRAPI)*, and wait for the IRE_INPUT_DONE event.

   Touch-tone input is always being collected and placed on the input queue. The IRE_INPUT event indicates that input has been placed on the input queue. The IRE_INPUT_DONE event indicates that the input on the input queue matches conditions specified by the input queue parameters.

   - IRP_INPUT_LEN
   - IRP_TT_PRETIME
   - IRP_TT_INTERTIME
   - IRP_INPUT_DELIM1
   - IRP_INPUT_DELIM2

   Chantest sets these input parameters during application initialization with its setTTParams() subroutine. After the playing of prompt is started, chantest returns to the main while loop to wait for the next event. If the caller waits for play to complete before entering any input, the IRE_PLAY_DONE event arrives when chantest's PlayDone flag is set to START_TIMER_WHEN_DONE. Chantest must start the touch-tone timer with *irStartTTTimer(3IRAPI)*. Chantest then returns to the main while loop to wait for the IRE_INPUT_DONE event:

```
case IRE_PLAY_DONE:
  switch(Chl[chan].PlayDone) {
      .  .  .
    case START_TIMER_WHEN_DONE:
      if (irStartTTTimer(cid) == IRR_FAIL)
          cleanup("irStartTimer Failed", cid);
      break;
    case INPUT_DONE_WHEN_DONE:
      input_done(cid, &Chl[chan].InputDoneEvent);
      break;
```

IRE_INPUT and IRE_INPUT_DONE events are also handled in the main while loop. The IRE_INPUT event is simply ignored since nothing needs to be done until the IRE_INPUT_DONE event arrives. The touch-tone timer does not have to be restarted before the IRE_INPUT_DONE event. The IRAPI automatically restarts the timer using the IRP_TT_INTERTIME parameter value between IRE_INPUT events.

If the IRE_INPUT_DONE arrives when the IRAPI library is still in the IRS_PLAYING state, processing of the IRE_INPUT_DONE event is delayed until the IRE_PLAY_DONE event arrives. This is done by setting the channel's PlayDone flag to INPUT_DONE_WHEN_DONE and saving the IRE_INPUT_DONE event structure. The IRE_INPUT event always precedes the IRE_PLAY_DONE event when speech is talked off. The library is not in the IRS_IDLE state until the IRE_PLAY_DONE event is received. (See above where the INPUT_DONE_WHEN_DONE value is used with the IRE_PLAY_DONE event.) This ensures that the IRAPI is in the IRS_IDLE state before the application continues. The IRAPI must be idle before more speech can be queued to play.

```
case IRE_INPUT_DONE:
  if ( irLibState(cid) == IRS_PLAYING ) {
      Chl[chan].PlayDone = INPUT_DONE_WHEN_DONE;
      Chl[chan].InputDoneEvent = ev;
  } else {
      input_done(cid, &ev);
  }
  break;

case IRE_INPUT:    /* Event is ignored */
  break;
```

The input_done() subroutine evaluates the modifiers from IRE_INPUT_DONE:

```
void input_done(channel_id cid, ir_event_t *evPtr)
{

    int chan = irCid2Chan(cid);

    switch(evPtr->event_mod1) {
        case IREM_INPUT_LENGTH:
        case IREM_INPUT_DELIM:
            Chl[chan].RetryCount = 0;
           /* play back touch tones to caller */
            play_tt(cid);
            break;
        case IREM_TT_PRE:
        case IREM_TT_INTER:
            if(Chl[chan].RetryCount++ >= 3) {

             /*
              * 3 tries with no input.  Abort transaction.
              */
              if ( irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
                irFPlay(cid, 0, "/speech/aborted") == IRR_FAIL ||
                irFPlay(cid, 0, "/speech/bye") == IRR_FAIL ||
                irEnd(cid, 0, 0) == IRR_FAIL  ) {
                  cleanup ("Error processing IRE_INPUT_DONE", cid);
                  return;
               }
                Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
            } else {
             /* play instructions again */
                playInstr(cid);
            }
            break;
        default:
            cleanup("Unexpected IRE_INPUT_DONE event modifier", cid);
            break;
    }
}
```

Input is removed from the input queue with *irGetInput(3IRAPI)* and played back to the caller in chantest's play_tt() subroutine:

```
void play_tt(channel_id cid)
{
    int len;
    char buf[INPUT_LEN + 1];
    char *bufPtr;

    if ( (len = irGetInput(cid,buf,INPUT_LEN)) == IRR_FAIL ) {
        cleanup("irGetInput Failed in play_tt", cid);
        return;
    }
    buf[len] = 0;

    if (irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_IGNORE) == IRR_FAIL ){
        cleanup("irSetEvent failed in play_tt", cid);
        return;
    }

    for(bufPtr = &buf[0]; *bufPtr != 0; bufPtr++) {
        switch(*bufPtr) {
          case '0':
            (void) irFPlay(cid, 0, "/speech/n.0");
            break;
          case '1':
            (void) irFPlay(cid, 0, "/speech/n.1");
            break;
            .
            .
            .
          case '9':
            (void) irFPlay(cid, 0, "/speech/n.9");
            break;
        }
    }
    if(strcmp(buf, "0000") == 0) {
        if (irFPlay(cid, 0, "/speech/bye") == IRR_FAIL) {
            cleanup ("Error on irFPlay", cid);
            return;
        }
        Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
    } else {
        Chl[chan].PlayDone = REPROMPT_WHEN_DONE;
    }
    if ( irLibState(cid) != IRS_PLAY_QUEUED ) {
        reprompt(cid);
        return;
    }
    if (irEnd(cid, 0, 0) < 0) {
       cleanup  ("Error on irEnd", cid);
    }
}
```

Notice that the play_tt() subroutine shown above plays back caller input after using *irSetEvent(3IRAPI)* to set the IRE_INPUT event to IRF_NOTIFY only, thereby clearing the IRF_ITR flag. This disables ''talkoff'' so that the caller may not interrupt this play. A check is done to see if the caller entered 4 zeros. If so, a ''goodbye'' message is played and the channel's PlayDone flag is set to disconnect when the IRE_PLAY_DONE event for that message arrives.

4. Disconnect with *irDisconnect(3IRAPI)*, wait for the IRE_DISCONNECT_DONE event, and terminate the application.

Application Termination

After disconnect, the application should return the channel to the system default owner with *irDeinit(3IRAPI)*. Once *irDeinit()* has been called, the cid value originally obtained for the channel through *irInit(3IRAPI)* or *irInitGroup(3IRAPI)* is invalid and should no longer be used.

The application should also reset or deallocate any application specific data that it uses.

Chantest disconnects using *irDisconnect(3IRAPI)* on the input ''0000'' or upon multiple input timeouts [see the play_tt() and input_done() subroutines]. When *irDisconnect(3IRAPI)* is called, chantest must wait for an IRE_DISCONNECT_DONE event before releasing the channel with *irDeinit(3IRAPI)*. The following code fragment shows how chantest disconnects after a ''goodbye'' announcement has finished playing (when its PlayDone flag has been set to DISCONNECT_WHEN_DONE):

```
 case IRE_PLAY_DONE:
switch(Chl[chan].PlayDone) {
      .  .  .
    case DISCONNECT_WHEN_DONE:
      if (irDisconnect(cid, 0) < 0) {
          cleanup("Error on irDisconnect", cid);
      }
      break;
      .  .  .
  }
  break;

 case IRE_DISCONNECT_DONE:
  if (irDeinit(cid) < 0) {
       cleanup("Error on irDeinit", cid);
  }
  break;
```

Once *irDeinit(3IRAPI)* is called, the channel is returned to the default owner (the AD process) and the associated cid obtained through *irInit(3IRAPI)* becomes invalid.

Applications must also handle network events such as hang-up. If the caller hangs up before the application terminates, an IRE_DISCONNECT or IRE_WINK event is generated. Applications should set these events during application initialization to interrupt voice play when they occur (see chantest's setEvents() subroutine). When the hang-up events occur, the application should call *irDisconnect(3IRAPI)* to disconnect. If the IRAPI library is still in the IRS_PLAYING state when the IRE_DISCONNECT or IRE_WINK event arrives, the disconnect processing should be delayed until the IRE_PLAY_DONE event arrives. The following code waits for the IRE_PLAY_DONE event since *irDisconnect(3IRAPI)* cannot be called until the channel is idle.

```
case IRE_DISCONNECT:
case IRE_WINK:
  if (irLibState(cid) == IRS_PLAYING) {
      Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
  } else if (irDisconnect(cid, 0) < 0) {
      cleanup ("Error on irDisconnect", cid);
  }
  break;
```

An alternative to waiting for the IRE_PLAY_DONE event is to use *irDeinit(3IRAPI)*.

Process Termination

IRAPI processes should *irDisconnect(3IRAPI)* and *irDeinit(3IRAPI)* any channels they are using. If the process should terminate ungracefully, however, the system returns the channel and any system resources being used by it to their initial state.

The chantest application is designed to be a permanent process. It starts when the voice system is started and does not terminate until the voice system is stopped. A transient process, designed to terminate when the application terminates, would only need to use the *exit(2)* UNIX system call.

Run-Time Services

The following sections detail the IRAPI run-time services. The run-time services are grouped under the following headings:

- Channel management
- Event and interrupt management
- Call profile
- Voice operations
- Telephony support
- Timeslot management
- Speech file access
- Speech recognition
- Resource management
- Text-to-Speech (TTS)
- Platform management

Channel Management

Channel management involves operations that affect the ownership of channels and the passing of ownership from one channel to another. Except for very short intervals when the channel ownership is being transferred from one channel to another, all channels must be owned by some process. Only the channel owner may affect the behavior or the activities of a channel. Processes may compete for channel ownership and may wait for channel ownership as they would any other resources. Processes have some control over when channels may be taken away from them; however, maintenance processes may remove channels forcibly from other processes.

The *channel_id* or *cid* discussed throughout this section is used by the IRAPI to associate a channel to its pertinent information. Specific functions convert channel numbers to *channel_id*'s and *channel_id*'s to channel numbers [see *irChan2Cid(3IRAPI)* and *irCid2Chan(3IRAPI)* in Appendix A, ''Reference Material''].

Default Ownership

Since all channels must be owned by some process, when no process takes ownership of a channel, the default owner becomes the channel owner. All in-service channels are owned by their default owner; the *default* default owner is AD. Alternate default owners can be created (discussed later in this section).

Primarily, the default owner handles IRE_NEWCALL events. The IRE_NEWCALL event indicates that a new call has arrived. Assuming AD is the default owner, the AD tables are queried based on the channel number, dialed number identification number (DNIS), or automatic number identification (ANI) values. Through this query, a process to which ownership of the channel should be given is identified and the channel is *irExec(3IRAPI)*'d to that process. After being *irExec(3IRAPI)*'d, the IRP_SERVICE_NAME parameter is set to the value indicated during service definition [see **defService** in the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230].

The default owner holds all idle, in-service channels. An idle channel is any channel whose service state is IRD_INACTIVE [see *irServiceState(3IRAPI)*] and whose library state is IRS_IDLE [see *irLibState(3IRAPI)*]. Service and library states are described in more detail later in this chapter. When processes request channels owned by the default owner, the channel is typically relinquished upon request. Control over relinquishing idle channels is enabled through the IRP_CHAN_NEGOTIATION parameter (described in more detail later in this chapter).

All out-of-service channels are owned by MTC_PROXY rather than the default owner. MTC_PROXY takes ownership of out-of-service channels and does not relinquish ownership of the channel unless instructed to do so by the MTC process. MTC_PROXY may also forcibly seize channels from AD or other owners when in-service channels are being taken out-of-service.

When a process relinquishes ownership of a channel and there are no other processes pending for ownership of that channel, channel ownership is returned to the default owner. The default owner receives the IRE_DEFOWN event when channel ownership is returned. After receiving the IRE_DEFOWN event, the default owner should take ownership of the channel via *irInit(3IRAPI)*.

Execing Applications on Channels

A process ''exec's'' another process on a channel via an *irExec(3IRAPI)* function. Channels may be *irExec*'d to permanent or transient processes. Permanent process are typically multi-channel applications run from *inittab*. Transient processes are *exec(2)*'d by the *irExec* function and typically *exit(2)* when the call, or their portion of it, completes.

Ownership of the channel is relinquished by the current owner with an *irExec*. Once an *irExec* function is called, the calling process must stop using the *channel_id* . As was discussed earlier, AD *irExec*'s channels based on application assignments.

Channel ownership is made available immediately to the *irExec*'d application. After receiving the IRE_EXEC event, the channel should be initialized immediately via *irInit(3IRAPI)*.

Parameters may be used to allow the *child* process to run in a context similar to the *parent* process. Also, the following parameters may be used to allow the parent process pass data to the child process:

- IRP_EXEC_BUF is a data buffer that is not interpreted by the IRAPI. A parent process sets this buffer and its length through *irExec(3IRAPI)* arguments. The child application then accesses the data via *irGetParamStr(3IRAPI)*.
- The parameter IRP_REGISTER allows data to be passed similar to IRP_EXEC_BUF; however, there is not a length parameter associated with IRP_REGISTER and parent processes must set explicitly the parameter via *irSetParamStr(3IRAPI)*.

Many IRAPI parameters and parameter values are saved across *irExec* boundaries. Any resources explicitly allocated or restricted via *irReserveResource(3IRAPI)* and *irRestrictResource(3IRAPI)*, respectively, are saved across *irExec* boundaries. Half bridges *[irHBridge(3IRAPI)*] remain active across *irExec* boundaries. Echo cancellation [*irEcho(3IRAPI)*] remains on. See *IrPARAMETERS(4IRAPI)* for a complete list of the parameters.

Execing TSM Script Applications

The following parallels exist between the IRAPI and TSM scripts:

- IRAPI IRP_EXEC_BUF parameter and the TSM script argument X.0

  Any TSM script *irExec*'d from an IRAPI application has access to data passed via the IRP_EXEC_BUF parameter through the X.0 code.

- IRAPI IRP_REGISTER parameter and the TSM script registers r.0 through r.15

  The TSM script registers are preloaded with the values found in IRP_REGISTER, assuming that the data for this parameter is arranged as a block of 16 continuous integer values.

Refer to Chapter 3, "Script Instructions," of the *Intuity CONVERSANT VIS V5.0 Application Development*, 585-310-227, for information on argument types.

The following example shows how a tas application is started from an IRAPI process. The registers are set with the data supplied through the *register* argument and the *buf* and *buf_len* arguments set the exec buffer. Note that IRP_EXEC_BUF and IRP_EXEC_BUF length are set automatically by the *irExecp* function. The define symbol ''TSM'' is defined in **/att/include/mesg.h**.

```
int exec_tas(channel_id cid, const int *register, const char *buf,
  int buf_len, const char *service)
{
    (void) irSetParamStr(cid, IRP_REGISTER, register);
    if ( irExecp(cid, service, TSM, buf, buf_len) == IRR_FAIL ) {
       return(0);
    }
    return(1);
}
```

Gaining Ownership of a Channel

The *irInit(3IRAPI)* and *irInitGroup(3IRAPI)* functions support requests to gain ownership of a channel. *irInit(3IRAPI)* attempts to gain ownership of a particular channel, while *irInitGroup(3IRAPI)* attempts to gain ownership of some channel in a group. The concept of channel groups is unchanged from prior releases and groups are administered in the same way (refer to Chapter 3, ''Configuration Management,'' of *Intuity CONVERSANT VIS V5.0 Operations*, 585-310-550). However, the following special groups are defined with the IRAPI:

- IRD_REAL_CHANS — Includes all channels associated with a physical network connection
- IRD_VIRTUAL_CHANS — Includes all channels defined through the IRP_VIRTUAL_CHANS global parameter [see *irAPI.rc(4IRAPI)*]. Virtual channels allow applications to run on channels that are not associated with a physical network port. Virtual channels support a subset of IRAPI operations.

Gaining ownership of a channel depends on current ownership and activity. If a channel was just *irExec*'d to an application, the channel is, in effect, *unowned* for a short period of time. The processes receiving the IRE_EXEC event (the *irExec*'d process) should take ownership of the channel immediately via *irInit(3IRAPI)*. The IRAPI does not allow the channel to go unowned for more than 5 seconds; if the target process does not take ownership, ownership returns to the default owner. If, after receiving an IRE_EXEC event, a process immediately calls *irInit(3IRAPI)* on the channel, channel ownership is granted immediately [as indicated by a return code of IRR_OK from *irInit(3IRAPI)*].

A process may attempt to ''seize'' ownership of a channel. For example, processes may seize channels as a result of a request to start an outbound call. *irInit(3IRAPI)* should be used to seize ownership of a particular channel and *irInitGroup(3IRAPI)* should be used to seize ownership of any idle channel from a particular group.

When seizing a channel, a process must wait for channel ownership since negotiations must be carried out with the current channel owner. Therefore, a request for channel ownership not made as a result of an IRE_EXEC event returns either IRR_FAIL or IRR_PENDING.

A channel requested from another process is removed from the current owner if both of the following are true:

1. IRP_CHAN_NEGOTIATION is set to IRD_ALLOW
2. The channel is idle; that is, the service state is IRD_INACTIVE and the library state is IRS_IDLE

When a channel is taken away from a process in this manner, an IRE_CHAN_REMOVED event is sent to the previous owner.

If IRP_CHAN_NEGOTIATION is set to IRD_CONDITIONAL, the current owner receives the IRE_CHAN_REQUESTED event. The current owner may ignore the event or release the channel via *irDeinit(3IRAPI)*.

AD sets IRP_CHAN_NEGOTIATION to IRD_ALLOW; therefore, unless AD is dispatching a call on the channel (that is, the service state is not IRD_INACTIVE), AD gives up ownership of channels immediately upon request.

A process secures ownership of a channel when:

- *irInit* returns IRR_OK
- The IRE_CHAN_GRANT event is received following a return of IRR_PENDING from *irInit*

IRE_CHAN_DENY is returned if the current owner does not release the channel after an *irInit(3IRAPI)*, or if no channel in the group specified with *irInitGroup(3IRAPI)* is released before the timeout specified in *return_mode*.

The *outcalling* version of chantest (chantest_oc.h) uses *irInit(3IRAPI)* to seize an idle channel. Refer to Appendix B, "Sample Applications."

Relinquishing Ownership of a Channel

A process relinquishes ownership of a channel by one of the following:

- Calling *irDeinit(3IRAPI)* — Returns channel ownership to the default owner or a pending owner. This is the recommended and preferred method to relinquish ownership. It allows the library a chance to idle the channel and return it gracefully to the default owner, or to hand channel ownership to a pending process. The system considers process termination of channel owners to be an error and a system error is logged.
- Calling *exit(2)* or some other process termination, such as dumping core — Returns channel ownership to the default owner or a pending owner. By definition, transient processes *exit(2)* after successfully releasing a channel via *irDeinit(3IRAPI)*; however, if the channel is not in the IRS_IDLE state, the IRAPI, running in the context of the current channel owner, must idle the channel. Idling a channel may require asynchronous

communication with other system processes. The process must wait for the IRE_DEINIT_DONE event before exiting. The code fragment below shows the expected behavior of transient processes. This example assumes that after the application completes a play, it releases the channel and exits. Note that IRE_DEINIT_DONE is disabled by default and must be explicitly enabled.

- Calling *irExec(3IRAPI)* to pass ownership to another process — Renders the *channel_id* invalid and removes channel ownership from the calling process after a successful return from *irExec(3IRAPI).* Refer to ''Execing Applications on Channels'' earlier in this chapter for additional information.
- Having ownership forcibly removed — Generates the IRE_CHAN_REMOVED event and renders the *channel_id* invalid

```
irWCheck(&ev) {
    switch(ev.event_id) {

        ...


      IRE_PLAY_DONE:
        (void) irSetEvent(ev.cod, IRE_DEINIT_DONE, IRF_NOTIFY);
        (void) irDeinit(ev.cid);
        break;

      IRE_DEINIT_DONE:
        exit(0);

        ...

    }
}
```

Library States

The IRAPI maintains state information about channel activities. These *library states* prevent applications from overloading telephony hardware and eliminate ambiguities in the library. For example, an idle channel is in the IRS_IDLE library state, and a playing channel is in the IRS_PLAYING library state. Most states indicate the activity currently active on the channel. There are also several state modifiers including:

- IRS_PENDING — Indicates that some activity is waiting until resources are available. The activity cannot be started until resources are available, at which time the IRS_PENDING modifier is cleared.
- IRS_QUEUED — Indicates that an activity has been queued but not yet started
- IRS_DEINITING — Indicates that a channel is being deinitialized. A *channel_id* in this state can *not* be used.

The *irLibState(3IRAPI)* function is used to determine the library state of a channel. The *irSName(3IRAPI)* function returns a character string containing the name of the states and/or modifiers of the library state passed as an argument.

Library states are useful for resolving ambiguities about the ordering of events, such and IRE_INPUT and IRE_PLAY_DONE, or for debugging applications. Applications should not use the library state to drive behavior. Application behavior should be driven by the events that result from asynchronous activities.

Event and Interrupt Management

IRAPI applications are event driven. This means that the application must respond properly to completion events that result from commands initiated by the IRAPI application (such as a speech play request) or miscellaneous externally induced events (such as an incoming telephone call). See *irEVENTS(4IRAPI)* for a description of all possible events.

By default, the library notifies the application of most events. The application may request that it be notified of all the events. Some events can be *masked* meaning that an application may ask not to be informed of the event (that is, the event is to be ignored). Some of the more important events are *non-maskable* meaning that the application *must* be informed about the event. Generally an event is *non-maskable* if the application would likely encounter state transition errors by trying to ignore the event.

By properly responding to the events as they occur, an application seldom needs to check the library or service states that are maintained by the IRAPI library. Applications may occasionally need to maintain a small amount of application state information in order to properly handle the events that occur.

The IRAPI library allows an application to interrupt activities with events. Interrupts refer to the premature stopping of some IRAPI activity such as playing of speech. For example, an application may want to interrupt speech after receiving a touch tone so the caller does not have to listen to the entire prompt.

Controlling Events

The action taken by the library in response to an event may be modified by the routines described in *irEvent(3IRAPI)*.

- *irSetEvent(3IRAPI)* may be used to control the action taken for a library event. This routine may be used to ignore an event, to enable the reporting of the event, or to have the event automatically interrupt voice or telephony activity on the channel.

  The *action* argument to *irSetEvent* is an ORed result of some of the following values:

  — IRF_IGNORE — Ignore the event

  — IRF_NOTIFY — Notify on event

  — IRF_PLAYINTR — Interrupt playing

  — IRF_RECINTR — Interrupt recording

  — IRF_SAYINTR — Interrupt saying (Text-to-Speech)

  — IRF_CALLINTR — Interrupt calling

- *irGetEvent(3IRAPI)* can be used to return the current control action for an event.
- *irInitEvents(3IRAPI)* can be used to reset event control actions for all events to the default actions. This function is called automatically by *irDeinit(3IRAPI)* and also can be called explicitly by the application.

Detecting Library Events

The following routines are used to detect library events:

- *irWait(3IRAPI)* waits for an event to occur (but not return the event).
- *irCheck(3IRAPI)* gets the next event (or IRE_NULL as an indication that there are no more events).
- *irWCheck(3IRAPI)* waits for an event and then returns the first event to be processed by the application. *irWCheck()* does not return until there is an event to report. This function combines the actions of *irWait()* and *irCheck()* and is suitable for most applications.

  NOTE:
  A few applications may need to call *irCheck()* directly if they need to be able to detect that there are no pending events and then take some action other than calling *irWait()* to wait for the next event.

The following code fragment shows an example of *irWCheck()*:

```
while (irWCheck(&ev) != IRR_FAIL) {

    cid = ev.cid;
    chan = irCid2Chan(cid);

    switch(ev.event_id) {

      case IRE_EXEC:

        ...

      case IRE_PLAY_DONE:

        ...

    }
}
```

Handling Events in a Timely Manner

An IRAPI application must handle events in a timely manner to prevent problems from occurring. Network interface timing errors may occur, or callers may hear speech breaks or other undesirable behavior.

An IRAPI application must call *irWCheck* (or *irWait* and *irCheck*) frequently to avoid these problems. This is especially true for permanent IRAPI applications that are handling multiple channels simultaneously. A permanent IRAPI application must not allow itself to be blocked while waiting for an input/output (I/O) or other request that could take more than a few milliseconds to complete. For example, it is unwise to write an IRAPI application that is blocked while waiting for the response to a complex database query that might take seconds to complete. A separate data interface process (DIP) may be necessary in this case.

Preferably, IRAPI applications should be written so that the only blocking occurs within *irWait* or *irWCheck*. Internally, *irWait* uses the UNIX system call **msgrcv** to wait for a message that might represent an IRAPI event. *irCheck()* also calls **msgrcv** (with a no-wait flag) if there are no pending events.

Polling with Streams Devices

irWait is compatible with UNIX signals in that SIGPOLL can be used with stream-oriented devices. The application should make non-blocking requests for these devices and use I_SETSIG. Then, SIGPOLL interrupts *irWait* and the SIGPOLL interrupt handling routine can handle the stream-oriented events.

```
void
handle_poll (int signal_no)
{
       /*  handle your input here, taking care not to sleep  */
}

main ()
{
       sigset (SIGPOLL, handle_poll);
       ioctl (0, I_SETSIG, S_INPUT);

       while (1) {
              while (irWCheck(&ev) != IRE_NULL) {
                     switch (ev.event_id) {

                            ...

                     }
              }
       }
}
```

Polling for Other Input Using IRAPI Timers

If an application simply needs a frequent stimulus to poll for the handling of non-IRAPI activity, the application can set a process-oriented timer to generate an IRE_CLOCK event at regular intervals with ten millisecond granularity. Refer to the following code fragment for an implementation example

```
main ()
{
       int    tag=0xfeed;

       /*  set a repeating timer for 2 seconds  */
       irStartPTimer (2000, 1, tag);
       while (1) {
              while (irWCheck(&ev) != IRE_NULL) {
                     switch (ev.event_id) {
                     case IRE_CLOCK:
                            /*  handle your other activity here,
                             *  taking care not to sleep  */

                            ...

                     }
              }
       }
}
```

Polling for Other Input with SIGALRM

The UNIX *alarm* system call can be used to interrupt *irWait* at fairly regular intervals to handle other activity input. The use of *alarm* allows less control on the accuracy of the polling rate than using IRE_CLOCK in the previous example because *alarm* generates an interrupt at the given time with one second granularity.

```
void
handle_alarm (int signal_no)
{
       /*  handle your input here, taking care not to sleep  */
       alarm (2);
}


main ()
{
       sigset (SIGALRM, handle_alarm);
       /*  set a repeating timer for 2 seconds  */
       alarm (2);
       while (1) {
              while (irWCheck(&ev) != IRE_NULL) {
                     switch (ev.event_id) {

                            ...

                     }
              }
       }
}
```

Polling for IRAPI Activity

As a last resort, an IRAPI application can block on other activity (only when necessary) and poll the IRAPI library as frequently as possible.

```
main ()
{
       while (1) {
              /*  wait on and handle your input here,
               *  making sure to enter this loop frequently.  */

              while (irCheck(&ev) != IRE_NULL) {
                     switch (ev.event_id) {

                            ...

                     }
              }
       }
}
```

IRAPI Event Tags

IRAPI functions that initiate speech or telephony activity which result in a subsequent completion event have a *tag* argument. This *tag* can be used for whatever purpose the application chooses. The same *tag* value is returned in the *tag* field of the resulting completion event [see *IrEVENTS(4IRAPI)].*

The *tag* value is not used internally by the IRAPI library and may contain any integer value that the application chooses. The *tag* can be used to resolve any ambiguity on event handling. The application may choose to use the *tag* for any of the following:

- Application state information (for example, indication of next action to take)
- Resolving race conditions (for example, sequence numbers)
- Table lookup (for example, matching an IRE_CHAN_GRANT event with the corresponding *irInitGroup* function)

Call Profile

The call profile is a set of data elements that provide a context that determines the behavior of some functions, internal system event processing, and application dispatching.

The call profile is divided into two groups:

- *Parameters* affect the operation of functions and event processing.
- *Information elements* allow information to be transmitted from the telephone network to the IRAPI and vice versa.

Parameters

Parameters are used to affect the behavior of or set options on functions. Once parameters are set initially for some application, they need not be re-specified for a function each and every time it is used. Therefore, for many functions, parameters are used instead of function arguments.

There are two types of parameters, channel specific and global. Channel specific parameters must be set and queried for each individual channel. Global parameters are set system wide; they are also read only, and are not part of the call profile since they are not specific to a channel or a call.

Channel-Specific Parameters

Channel specific parameters, or parameters, for the purposes of this section, are defined in *IrPARAMETERS(4IRAPI)*. Each parameter has several attributes defined as follows:

type	Indicates an integer or string type
size	Indicates the size, in bytes, required to contain the parameter value. All integer type parameters occupy *sizeof(int)* bytes; string type parameters vary in size.
save-on-exec	Indicates the parameter value persists across a call to *irExec(3IRAPI)*. Parameters that are saved across an *irExec(3IRAPI)* provide a communication path from parent to child process and allow a child process to execute in a context similar to the parent's with respect to IRAPI functions.

Parameter values may be retrieved via *irGetParam(3IRAPI)* and *irGetParamStr(3IRAPI)* for integer and string type parameters respectively. Parameter values may be set via *irSetParamStr(3IRAPI)* and *irSetParamStr(3IRAPI)* for integer and string type parameters, respectively.

When a channel is released via *irDeinit(3IRAPI)*, all parameters are reset to their default values. The default values are listed in *IrPARAMETERS(4IRAPI)*. The *irInitParam(3IRAPI)* and *irInitAllParams(3IRAPI)* functions are also provided to set a single or all channel parameters to its/their default values. Most default values are not affected by the application environment or system administration settings; however, some are. For instance, the default value for IRP_FLASH_DURATION is taken from the values specified by the switch integration settings (see Chapter 6, ''Switch Interface Administration,'' of *Intuity CONVERSANT VIS V5.0 Operations*, 585-310-550). Furthermore, its value is meaningless for certain telephony types.

Since channels are released to AD through *irDeinit(3IRAPI)* and AD does not change the parameters' default settings, any channel *irExec(3IRAPI)*'d from AD on an IRE_NEWCALL event has its parameters set to their default values.

NOTE:
Getting and setting string type parameters should be done with caution. *irGetParamStr(3IRAPI)* copies exactly the number of bytes specified. It does not interpret NULL characters nor does it NULL terminate strings. Applications are required to NULL terminate strings if the parameters containing them are as large as the string. *irSetParamStr(3IRAPI)* also does not interpret NULL characters and always copies the full parameter size out of the program space. Therefore, source arguments to *irSetParamStr* should point to sufficiently large areas of memory. The non-interpretation of NULL characters allow string type parameters to be used for both character string and buffer data types.

The chantest.c application in Appendix B, ''Sample Applications,'' shows getting and setting string-type parameters.

Global Parameters

Global parameters are used to set system wide options that affect the behavior of some IRAPI functions. Global parameters are set or modified by manually editing the **/vs/data/irAPI.rc** file. Global parameters are read-only from IRAPI applications. The *irGetGlobalParam(3IRAPI)* and *irGetGlobalParamStr(3IRAPI)* functions provide access to global parameters for integer and string type parameters, respectively. See *irAPI.rc(4IRAPI)* for global parameters descriptions and their default settings.

Information Elements

Information elements allow some network connections to describe incoming calls to the voice system. They also allow the voice system to describe outgoing calls to the network.

- Set by incoming calls: IRD_ANI, IRD_DNIS, IRD_REDIRECTING, and IRD_INBOUND_SERVICE.
- Set by application to describe outbound calls: IRD_SERVICE_TYPE, IRD_BEARER_CAP, and IRD_OUTBOUND_ANI.

The IRAPI interface to information elements is similar to the channel parameter interface, including integer and string types, and supported through the following functions:

- *irGetIE(3IRAPI)* — Gets an integer type information element
- *irGetIEs(3IRAPI)* — Gets a string type information element
- *irSetIE(3IRAPI)* — Sets an integer type information element
- *irSetIEs(3IRAPI)* — Sets a string type information element

See *irIE(3IRAPI)* in Appendix A, ‘‘Reference Material,’’ for detailed information on these functions.

Voice Operations

This section discusses how to perform voice operations using IRAPI functions. These functions are used to play pre-recorded speech and to record speech from the caller.

Speech Queuing

Voice coded speech data may be placed on the channel play queue from a file or an internal buffer. Play commences when the *irEnd(3IRAPI)* function is called. Coded speech [*irPlay(3IRAPI)* functions] and TTS [*irSay(3IRAPI)* functions] queuing or playing cannot be mixed. A call to *irEnd()* must be used between queuing different types of speech (voice or TTS). Play of one type of speech must be stopped before another type is queued for play.

Pre-recorded speech may be queued for play with any combination of the *irFPlay(3IRAPI)* or *irLP(3IRAPI)* functions. Speech stored in a voice file may be queued for play with the *irFPlay()* function containing the UNIX path name of the voice file. *irFPlay()* queues the entire contents of the file given for play. A portion of a file may be queued for playing by obtaining a voice file descriptor from *irOpen()* function and using it with the *irPlay()* function. The voice file descriptor may be positioned at any point in the file using *irLSeek(3IRAPI)* and passed to *irPlay()* with a length specification (in milliseconds). This queues the portion of the open file for play. Speech stored in an internal buffer may be queued for play with the *irBPlay()* function.

Language playback routines in *irLP(3IRAPI)* may be mixed with the *irPlay(3IRAPI)* routines to queue speech necessary for playing numbers and characters from the standard speech package. The *irSpeakChar()* function may be used to queue a character string for play. ASCII characters that do not have corresponding phrases in the standard speech package are skipped. The *irSpeakNum()* function may be used to play whole numbers. It does not support speaking numbers in the billions and trillions because most of these numbers do not fit into an integer variable. These functions also support speaking numbers and character strings with rising, falling, or total inflections.

Speech Play and Control

The following functions are used to control the actual playing of queued speech.

- The *irEnd(3IRAPI)* function starts play of queued voice or TTS. The IRF_REMEMBER flag may be used with this function to allow a voice play request to be restarted with *irPlayResume(3IRAPI)*. (This flag is not valid for TTS play requests.) Once *irEnd()* is executed, play must complete before more voice or text may be queued. Play is complete when the application receives an IRE_PLAY_DONE event.
- The *irPlayResume(3IRAPI)* function resumes a "remembered" play request after it has stopped [when the IRF_REMEMBER flag is used with *irEnd()*].

NOTE:
Applications should handle the possible denial or delay of voice resource allocation when *irEnd()* is used. Depending on the value of the IRP_RESOURCE_RETURNMODE parameter [see *irPARAMETERS(4IRAPI)*], *irEnd()* or *irPlayResume()* may return IRR_FAIL or IRR_PENDING if the voice resource is not immediately available.

- The *irGetVCount(3IRAPI)* function obtains the amount of time that voice activity has taken place. The return value of *irGetVCount(3IRAPI)* is only valid after an IRE_PLAY_DONE, IRE_PLAY_PROG or IRE_RECORD_DONE event. Time is accumulated from the most recent call to *irEnd(3IRAPI)*, *irPlayResume(3IRAPI)*, or *irRecord(3IRAPI)*.
- The *irStop(3IRAPI)* function stops voice activity on a channel before normal completion. Play is stopped when the application receives an IRE_PLAY_DONE event.

Voice Recording

The following functions are used to control voice recording.

- The *irPhReserve(3IRAPI)* function reserves space in a voice file for a subsequent recording. The amount of space reserved is determined by the voice coding algorithm used (that is, the value of the IRP_RECORD_ALGO parameter) and the amount of time (in milliseconds) passed to the function.
- The *irRecord(3IRAPI)* function records speech into a voice file descriptor obtained from *irOpen(3IRAPI)*.
- The *irFRecord(3IRAPI)* function records directly to a given voice file. It opens the file passed to it, records the caller's voice into the file, and closes the file when recording terminates.
- The *irBRecord(3IRAPI)* function records directly into an internal buffer of IRD_SPEECH_BUF_SIZE bytes. This function generates an IRE_RECORD_BUF event to signal the application to process the contents of the buffer before waiting for another event. The data in the buffer is overwritten as each IRE_RECORD_BUF is received.
- The *irRecordResume(3IRAPI)* function resumes voice recording after it has been stopped. The recording must have been initiated with the ''remember'' flag set in a call to *irRecord(3IRAPI)* or *irFRecord(3IRAPI)*.
- The *irStop(3IRAPI)* function terminates recording. Recording is stopped when the application receives an IRE_RECORD_DONE event.

Telephony Support

The IRAPI library includes several functions that support telephony operations such as placing and receiving calls. These functions provide as much consistency as possible across the different types of telephony hardware. Unfortunately, not all telephony types are capable of supporting all features, and IRAPI applications may behave slightly differently for some telephony types.

The telephony functions allow applications to:

- Answer a call via *irAnswer(3IRAPI)*. (The *chantest* sample application in Appendix B provides an example).
- Place an outbound call via *irCall(3IRAPI)*.
- Flash the hook state of a channel via *irFlash(3IRAPI)*.
- Outdial via *irDial(3IRAPI)*. This is frequently done to pass data via dual tone multi-frequency (DTMF) tones or to dial a number after flashing the line.
- Disconnect a call via *irDisconnect(3IRAPI)*. (The *chantest* sample application in Appendix B provides an example).
- Control speech energy detection via *irStartSpeechED(3IRAPI)*, *irStopSpeechED(3IRAPI)*, and *irCheckSpeechED(3IRAPI)*.
- Control Call Classification Analysis (CCA) via *irStartCCA(3IRAPI)*, *irStopCCA(3IRAPI)*, and *irCheckCCA(3IRAPI)*,

Telephony Service States

In addition to using the IRAPI library state to track the telephony activity occurring on a channel, the IRAPI library tracks the service state of the channel. *irServiceState(3IRAPI)* returns the current service state of the channel. The possible service states include:

- IRD_INACTIVE — Idle or on-hook
- IRD_RINGING — Incoming call has been detected but not yet answered
- IRD_ACTIVE — Active or off-hook (channel is in use)
- IRD_CHAN_OOS — Out-of-service
- Special primary rate interface (PRI) and adjunct/switch application interface (ASAI) states

The possible service states are described in detail under *irServiceState(3IRAPI)* in Appendix A, ''Reference Material.'' By properly handling events as they occur, an application seldom needs to determine the library state or service state for the channel it is controlling.

Using irCall

An IRAPI application can place an outbound call by using *irCall(3IRAPI)*. Typically, an IRAPI application receives an IRE_EXTERNAL event requesting it to place a call or it is *irExec(3IRAPI)*'d when the application is to place a call. After the request to place a call has been made, the IRAPI application must gain ownership of a channel via *irInit(3IRAPI)* or *irInitGroup(3IRAPI)*. Once the channel has been granted to the IRAPI application, it then may place the call via *irCall*.

irCall supports multiple types of CCA to determine the disposition of an outbound call. The type of CCA identified by the IRP_OUTCALL_CCALEVEL is used automatically to determine the results of the call attempt.

irCall uses the following library parameters to control the behavior of the call attempt:

- IRP_RESOURCE_RETURNMODE — Determines blocking status on CCA resource allocation
- IRP_OUTCALL_CCALEVEL — Determines the CCA level (blind, simple, full)
- IRP_OUTCALL_ANSDET — Determines the type of answer detection to be used when Full CCA is being used
- IRP_OUTCALL_DIALTYPE — Determines the outbound dial type
- IRP_OUTCALL_MAXRINGS — Determines the maximum number of rings to be detected before returning an event indicating ''no answer''

The completed call disposition is reported via an IRE_CALL_DONE event. The event modifiers indicate the disposition (for example, answered, busy, etc). The IRE_CALL_DONE event indicates whether the call attempt via *irCall* was successful. If the event modifiers indicate that the call was successfully completed, the application proceeds to another activity on the channel (for example, initiate playing of speech via *irPlay*).

A process must always call *irDisconnect(3IRAPI)* or *irDeinit(3IRAPI)* if an IRE_CALL_DONE indicates an error or failure occurred. In general, the IRAPI leaves the channel in the IRD_ACTIVE service state even if the call attempt failed.

In addition to the IRE_CALL_DONE event, an application also may receive one or more IRE_CALL_PROG events. This event reports intermediate events before or after the IRAPI determines that the call attempt is complete.

Making chantest Support Outcalling

An IRAPI application placing outbound calls must know what channel to use, what telephone number to call, and what parameter values to use (if not using the defaults). There are many techniques available to initiate outcalling. A transient IRAPI application that is designed to handle a single channel might accept command line arguments such as the telephone number to call. A permanent IRAPI application that is designed to handle multiple channels may receive the equipment group (or specific channel), phone number, and parameter values via an exec buffer or via an IRE_EXTERNAL event. The example that follows uses the IRE_EXTERNAL event to pass the equipment group, phone number, and parameter values.

The code fragments that follow indicate changes to the *chantest* sample application that allow the new *chantest_oc* application to initiate a call as well as answer incoming calls. The complete code for the *chantest_oc* sample application is provided in Appendix B, "Sample Applications." Also, included in Appendix B is the *mkcall* sample application that uses *irPostEventQ* to pass the IRE_EXTERNAL message to *chantest_oc*.

In this particular example, the *mkcall* program expects the user to provide the equipment group number that should be used when initiating the call. *chantest_oc* then uses any available channel in that equipment group to initiate the call. It is a straightforward change to *mkcall* and *chantest_oc* to pass either a specific channel number or an equipment group.

The following is the message structure for passing the data from the *mkcall* transient process to the *chantest_oc* process. *mkcall* is a command line program that accepts the data values as command line arguments and passes them to *chantest_oc* via the following message structure.

```
struct CALLDATA {
    struct mbhdr header;
    int groupNo;
    int MaxRings;
    int CCAType;
    char Number[PH_NUM_LEN];
} *msgp;
```

The *chantest_oc* program must store the call request parameters for one or more outbound call requests until the channels are granted for placing the call. The following structure is used to store the requests.

```
struct PENDING_CALL {
    int pending_flag;
    struct CALLDATA call_request;
} pending_calls[MAX_PENDING];
```

The following shows new code that can be added to *chantest* to handle the IRE_EXTERNAL event. Note that the index into the preceding array of structures is used as the identifier for a specific request.

```
      case IRE_EXTERNAL:
        /* Save the outcalling request data and soft seize a
         * channel from the specified equipment group.
         */
        msgp = (struct CALLDATA *) ev.event_text;
        for (i = 0; i < MAX_PENDING; i++) {
            if(pending_calls[i].pending_flag == IRD_FALSE) {

                pending_calls[i].pending_flag = IRD_TRUE;
                pending_calls[i].call_request = *msgp;
                (void) seizeGroup(i);
                break;
            }
        }
        break;
```

The new *seizeGroup()* function initializes a channel and possibly starts the application.

```
int seizeGroup(int req_idx) {

    struct CALLDATA *callReqP;
    int ret;
    channel_id cid;

    /* Attempt to become channel owner for a channel in the equipment
    ** group identified by the request.
    */
    callReqP = &pending_calls[req_idx].call_request;

    ret = irInitGroup(callReqP->groupNo, &cid, 10000, req_idx);

    if (ret == IRR_FAIL) {
        irPError("Error on irInitGroup");
        return(-1);
    }

    if(ret == IRR_OK) {
        if ( setEvents(cid) < 0 || setTTParams(cid) < 0 ) {
            cleanup("Error setting events or parameters", cid);
            return(-1);
        }
        return(startOutcall(cid, req_idx));
    } else {
        return(0);      /* Chan request is pending */
    }
}
```

In almost all cases, *irInitGroup(3IRAPI)* returns IRR_PENDING; therefore, the application must wait for either IRE_CHAN_GRANT or IRE_CHAN_DENY. Note that *tag* passed to *irInitGroup* is being used to identify a specific request in the array of PENDING_CALL structures.

```
case IRE_CHAN_GRANT:
  if(setEvents(cid) < 0) {
      cleanup("Error on setEvents", cid);
      break;
  }
  if(setTTParams(cid) < 0) {
      cleanup ("Error on setTTParams", cid);
      break;
  }

  (void) startOutcall(cid, ev.tag);
  break;

case IRE_CHAN_DENY:
  (void) fprintf(stderr, "Denied ownership of channel");
  if ( ev.tag >= 0 && ev.tag <= MAX_PENDING) {
      pending_calls[ev.tag].pending_flag = IRD_FALSE;
  }
  break;
```

The *startOutcall()* function initiates the outbound call via *irCall(3IRAPI)*.

```
int startOutcall(channel_id cid, int req_idx)
{
    struct CALLDATA *callReqP;
    int chan = irCid2Chan(cid);

    if(req_idx < 0 || req_idx > MAX_PENDING) {
        cleanup ("Invalid call request index", cid);
        return (-1);
    }
    /* Set parameters for outcalling and call.
     */
    callReqP = &pending_calls[req_idx].call_request;

    if ( irSetParam(cid, IRP_OUTCALL_DIALTYPE, IRD_DIALTYPE_TT)
             == IRR_FAIL ||
      irSetParam(cid, IRP_OUTCALL_MAXRINGS, callReqP->MaxRings)
             == IRR_FAIL ||
      irSetParam(cid, IRP_OUTCALL_CCALEVEL, callReqP->CCAType)
             == IRR_FAIL ||
      irCall(cid, 1, callReqP->Number) == IRR_FAIL ) {
        cleanup("Failure in startOutcall", cid);
        return(-1);
    }
    /* Release this pending_call entry */
    pending_calls[req_idx].pending_flag = IRD_FALSE;

    return(0);
}
```

The IRE_CALL_DONE event indicates call disposition. The IRE_CALL_PROG event may be used by some applications to follow call progress.

```
          case IRE_CALL_DONE:
            switch (ev.event_mod1){
              case IREM_ANSWER_SUP:
              case IREM_ANSWER:
              case IREM_BLIND:
                startChanTst(cid);
                break;
              case IREM_NOANSWER:
              case IREM_HIDRY:
              case IREM_REORDER:
              case IREM_BUSY:
              case IREM_ERROR:
              default:
                if (irDisconnect(cid, 0) < 0) {
                    cleanup ("Error on irDisconnect", cid);
                }
                break;
            }
            break;
```

Using Call Classification Analysis

Call Classification Analysis (CCA) is typically started as needed when using the *irCall(3IRAPI)* function. An IRAPI application must not explicitly start CCA when initiating a call via *irCall* since that could interfere with the automatic use of CCA.

There are some situations, such as dialing after flash transfers, where application control of CCA is required. The *irStartCCA(3IRAPI)*, *irStopCCA(3IRAPI)* and *irCheckCCA(3IRAPI)* functions support flexible use of CCA in those situations.

By default, *irDial* behaves as if simple CCA has been selected. IRAPI applications may start Full CCA explicitly by setting the IRP_OUTCALL_CCALEVEL parameter to IRD_FULL_CCA and then calling *irStartCCA(3IRAPI)* to start Full CCA. Full CCA allows more accurate detection of the disposition of the call made by a flash and dial. Full CCA requires an SP card with the CCA function assigned (refer to Chapter 3, ‘‘Configuration Management,’’ of *Intuity CONVERSANT VIS V5.0 Operations*, 585-310-550).

The following parameters affect CCA:

- IRP_OUTCALL_CCALEVEL — Determines the CCA level (blind, simple, full); must be set to IRD_FULL_CCA to start Full CCA
- IRP_OUTCALL_ANSDET — Determines the type of answer detection with Full CCA
- IRP_OUTCALL_MAXRINGS — Determines the maximum number of rings to be generated before returning a ‘‘no answer’’ disposition

The IRE_CALL_PROG event reports the tones and other events detected after the *irDial* function has been called. The application must determine whether the IRE_DIAL_DONE event or the IRE_CALL_PROG event represents the final event (whether successful or not) resulting from the flash and dial operation.

Timeslot Management

This section describes the the IRAPI functions that control the timeslots on the TDM bus on the Intuity CONVERSANT VIS.

The TDM bus is used in the VIS to transfer speech data between T1, Tip/Ring (T/R), and SP circuit cards. There are 255 communication paths, called timeslots, on the TDM bus. To transfer speech data between cards, an IRAPI application must reserve a TDM timeslot. There are two timeslots pre-allocated (or reserved) for each channel on the system. One of these reserved timeslots is used by the channel to output the speech data it receives from the network, while the other is reserved for inputing any speech data sent by other cards across the network. The channel can have up to 7 input timeslots, including the pre-allocated timeslot. Six timeslot spaces remain to perform activities such as playing background speech, half-bridging, or monitoring.

Applications start a background play request for a channel by using the *irStartBGPlay(3IRAPI)* function with the speech file name to be used for the background play. The file is continuously replayed until it is terminated by *irStopBGPlay(3IRAPI)* or when the channel is released via *irDeinit(3IRAPI)*. The global parameter IRP_BACKGROUND_OVOL sets the background output volume system-wide. The IRP_BACKGROUND_OVOL is a percentage relative to the channel's OVOL setting. The default value for IRP_BACKGROUND_OVOL is 33. The channel parameter IRP_OUTPUT_BGGAIN applies a dB gain factor to the background volume on a per channel basis. The default value is unity gain.

The *irHBridge(3IRAPI)* function implements half-bridging by adding another channel's input timeslot to the controlling channel's output. In other words, if channel A half-bridges to channel B, the customer on channel A hears what is being said by the customer on channel B. The customer on channel B is unaware that channel A has half-bridged to channel B. To perform a full bridge, channel B then must use *irHBridge* to half-bridge to channel A. Then, the customer on channel A hears what the customer on channel B is saying and vice-versa. The application must coordinate the two half-bridges to perform a full-bridge.

A channel can *listen* to another channel's input and output by using the *irMonitor(3IRAPI)* function. The transmission of DTMF digits across the TDM timeslots can be enabled or disabled by setting the IRP_DTMF_MUTING parameter. Refer to *irPARAMETERS(4IRAPI)* in Appendix A, ''Reference Material,'' for a discussion on enabling and disabling the IRP_DTMF_MUTING parameter.

Both *irHBridge(3IRAPI)* and *irMonitor(3IRAPI)* take channels and cid's as arguments and determine what timeslots are involved based on what channels are involved.

The IRAPI also provides the following functions that allow applications to work on a timeslot basis:

- *irTSAlloc(3IRAPI)* — Allows timeslots to be allocated to a channel id
- *irTSFree(IRAPI)* — Allows timeslots to be freed from a channel id
- *irTSEnd* — Starts activity on a channel [similar to *irEnd(3IRAPI)]*. The ordinary play functions (*irFPlay(3IRAPI)*, etc.) are used to queue up a play request. This function is only supported with play requests.
- *irTSStop(3IRAPI)* — Stops activity on a channel [similar to *irTSStop(3IRAPI)*]. This function is only supported with play requests.
- *irTSControl(3IRAPI)* — Allows timeslots to be added to or removed from a channel's output and also allows an application to set the gain control on a particular timeslot

In the following example, the application allocates a timeslot, places the timeslot in its output, queues speech files TS_PLAY_FILE1 and TS_PLAY_FILE2, and then calls *irTSEnd* to play the queued speech on the allocated timeslot. The IRE_TS_DONE event is used to indicate the end of a timeslot activity. The *event_text* element of the event structure points to another event structure containing an IRE_PLAY_DONE event.

```
void start_ts_play(channel_id cid)
{
    int ts;  /* Time slot */

    if ( (ts = irTSAlloc(cid)) == IRR_FAIL
      || irTSControl(cid, ts, 0, IRD_ADD) == IRR_FAIL
      || irFPlay(cid, 0, TS_PLAY_FILE1) == IRR_FAIL
      || irFPlay(cid, 0, TS_PLAY_FILE2) == IRR_FAIL
      || irTSEnd(cid, 0, ts) == IRR_FAIL ) {
       irDeinit(cid);
         return;
    }
}
```

Speech File Access

The IRAPI provides the following facilities to support speech file access:

- Voice file operators — Access arbitrary sections of a voice file through a voice file descriptor
- Algorithm detection — Determine the coding algorithm type of a voice object. A voice object is a UNIX file or a program memory buffer containing speech data.
- Algorithm conversion — Convert voice objects from one coding type to another
- Byte to time conversion and vice versa — Convert byte to time and time to byte
- Talkfile phrases to UNIX files — Convert talkfile phrase id pairs to UNIX file names and vice versa

NOTE:
All voice objects are encoded with some *algorithm* type. The IRAPI supports those algorithm types described in *IrALGORITHMS(4IRAPI)*. The terms coding type or encoding type refer to the algorithm type.

Voice File Descriptors

Voice file descriptors are similar to UNIX file descriptors. They are used by IRAPI applications to position the voice file pointer to arbitrary points, measured in milliseconds, and then play or record from those positions.

A voice file descriptor is allocated via *irOpen(3IRAPI)* and released via *irClose(3IRAPI)*. Once successfully opened, a voice file pointer may be re-positioned with *irLSeek(3IRAPI)*, played via *irPlay(3IRAPI)*, or recorded via *irRecord(3IRAPI)*.

Unlike file descriptors, voice file descriptors are not positioned as data is played from or recorded to them. The application must position the voice file pointer. Since voice file pointers are positioned in milliseconds, the IRAPI application is neither required to know the algorithm type nor to perform time to byte conversions.

After receiving IRE_PLAY_DONE (for single voice file descriptor plays) or IRE_PPLAY_DONE (for multiple voice object plays), a program could use *irGetVCount(3IRAPI)* reset the voice pointer. The following example shows how *irGetVCount* could be used to reposition the voice file descriptor after play completes.

```
/* At some point in the program a vfd is played... */

irPlay(cid, tag, vfd, 10000);

. . .

/* From the event processing routine ... */
while ( irWCheck(&ev) != IRR_FAIL ) {

    . . .

    case IRE_PLAY_DONE:

    /* Assume that the event was due to the vfd played above
     * reset voice file pointer to where play was stopped.
     */

    irLSeek(vfd, irGetVCount(cid), SEEK_CUR );

    . . .
```

The *irVfd2FD(3IRAPI)* function returns a file descriptor with the voice file pointer set to the byte count equivalent of the given voice file pointer. This file pointer is positioned in milliseconds.

Voice File Positioning and Speech Headers

The voice system uses speech headers to indicate the algorithm type of a voice object. Speech headers are four bytes in length and occur every 400 bytes. The *irLSeek(3IRAPI)* function and the *count* argument to the *irPlay(3IRAPI)* function do not account for speech file headers so *exact* time positioning of the voice file pointer or play duration is not guaranteed.

The voice file headers may also pose a problem when playing from arbitrary points of a voice file if the sequence of speech objects are not all encoded with the same algorithm. This is due to the way the voice cards play speech. A voice card receives a stream of speech data from the system. When a speech header is encountered, it sets its signal processors to decode with a certain processor algorithm. A stream of data composed of many speech objects encoded in a variety of algorithms works if each unique speech object contains a speech header at the beginning of the file. Playing from a voice file descriptor position to some arbitrary point within the voice file can cause garbled speech if the first four bytes do not represent a speech header *and* if this speech is preceded by a voice object encoded with a different algorithm. To avoid this problem, you can prepend all arbitrarily positioned voice plays with a speech header using *irBPlay(3IRAPI)*. The following example illustrates this technique.

In this example, the array of unsigned shorts is set to contain the byte sequence 0xff, 0xaa, 0x<code_type>. The algorithm type of the voice file associated with a voice file descriptor is obtained via a call to *irGetAlgorithm(3IRAPI)*, The byte sequence 0xff, 0xaa instructs the voice card that the subsequent byte is the

algorithm in which the subsequent speech is encoded. The call to *irBPlay* guarantees that the voice data played from voice file descriptor *vfd* is prepended with a speech header of the proper type, thereby allowing the passing data to the voice card of a different algorithm than that currently being decoded by the voice card.

NOTE:
This technique might degrade performance on large channel count systems since, internally, an entire speech block and speech file is allocated for the buffer play (that is, each buffer play acts like a unique file play). You may want to play a speech file created at installation time that contains only the header. In this case, the file is shared across channels and is likely to be cached in the speech buffer cache.

```
int play_vfd_with_head(channel_id cid, vf_descriptor vfd, int count)
{
    unsigned short header[2];

    header[0] = 0xffaa;
    header[1] = ((unsigned short) irGetAlgorithm(vfd)) & 0xff;

    if ( irBPlay(cid, tag, header, 4) == IRR_FAIL ||
      irPlay(cid, tag, vfd, count) == IRR_FAIL ) {
       return(IRR_FAIL);
    }
    return(IRR_OK);
}
```

Algorithm Detection

The *irGetAlgorithm(3IRAPI)* function returns the algorithm type of the voice file associated with a voice file descriptor (see the previous example under ‘‘Voice File Positioning and Speech Headers’’). Algorithm detection functions also exist to determine the encoding type of a voice file given a UNIX path name [*irFGetAlgorithm(3IRAPI)*], or a buffer [*irBGetAlgorithm(3IRAPI)*].

Algorithm Conversion

The following functions support the conversion of voice objects from one algorithm type to another:

- *irConvertAlgorithm(3IRAPI)* — Convert the algorithm from one voice file descriptor to another. This function supports conversion of a block specified from an arbitrary position of arbitrary length.
- *irFConvertAlgorithm(3IRAPI)* — Convert the algorithm from one voice file into another
- *irBConvertAlgorithm(3IRAPI)* — Convert the algorithm from one program buffer to another. The application developer must size the target buffer to accommodate speech headers.

Voice buffer conversion functions are unique in that they are asynchronous but are not associated with a channel. This allows for two things:

1. Using these functions in processes not owning channels such as command line utilities
2. Using these functions in processes owning channels without requiring those processes to block on the conversion (due to their asynchronous nature)

They are also unique in that they are computationally expensive. The *conv* process performs the computations on behalf of the IRAPI process on the main CPU. For this reason, these functions are best limited to utility processes. Voice response transactions should *not* use conversion routines in real time.

Byte and Time Conversions

The *irByte2Time(3IRAPI)* and *irTime2Byte(3IRAPI)* functions convert byte counts to milliseconds and milliseconds to byte counts given some algorithm.

NOTE:
These functions do not account for speech headers.

Talkfile Phrase ID Mapping

The *irTF2File(3IRAPI)* and *irFile2TF(3IRAPI)* functions provide a mapping to and from old talkfile/phrase-id numbers and UNIX file names. Mapping depends on the IRP_SPEECHDIR global parameter setting. Assuming IRP_SPEECHDIR is set to **/home2/vfs/talkfiles**, the talkfile/phrase id pair (100,200) is mapped to the UNIX file name **/home2/vfs/talkfiles/100/200** via *irTF2File*. Assuming IRP_SPEECHDIR as above, the filename **/home2/vfs/talkfiles/100/200** is mapped to the talkfile/phrase-id pair (100,200) via *irFile2TF*. If an existing application has speech stored according to talkfile/phrase-id pairs, *irFPlay(3IRAPI)* and *irTF2File* may be used together as follows.

```
if ( irFPlay(cid, 0, irTF2File(talkfile, phrase_id)) == IRR_FAIL ) {
```

Speech-Oriented Commands

The Intuity CONVERSANT VIS V5.0 stores speech in standard UNIX files. This allows the use of standard UNIX commands to manage speech data.

Prior to the Intuity CONVERSANT VIS V5.0 release, speech data was stored in a raw slice. Access to the raw slice was provided through a set of utility functions. While many of these utilities are directly replaceable with UNIX commands, most are still supported for the following reasons:

1. Backward compatibility with old administrative procedures
2. Access to speech data using the talkfile/phrase_id schema
3. Maintenance of voice system specific functionality not present in the standard UNIX commands. For example, the **list** command reports the time duration, coding type and phrase name of each file.

The following table lists the Intuity CONVERSANT VIS voice file system commands, the UNIX equivalents, their status, and a brief description of the command's function. For additional information on the voice commands, refer to *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230. For additional information on the UNIX commands, refer to the *UNIX SRV4.2 Command Reference*.

Table 3-1. Voice File System Commands

Voice Command	Status	Description	UNIX Analogue
vdf	**supported**	Find used/usable space in filesystem	df
copy	**supported**	Copy speech phrases	cp, cpio
audit	**obsolete**	Audit a filesystem	fsck
buildfs	**obsolete**	Build a raw slice speech filesystem	mkfs
spch_header	**supported**	Add speech headers to a phrase	-
ls	**supported**	List phrases in a speech slice	list
rm	**supported**	Remove (erase) speech phrases	erase
add	**supported**	Add speech to a filesystem	cp, cpio
spsav	**supported**	Save speech to a tape	-
spres	**supported**	Restore speech from a tape	-

The voice file specific commands use the IRP_SPEECHDIR global parameter to determine the locations of the relevant voice files.

Speech Recognition

Speech recognition provides caller input to an application. Like touch-tone input, speech recognition applications use the input queue to retrieve recognized data. The IRE_INPUT_DONE event indicates when something has been recognized or the recognition timed out. A separate recognition timer is available to support timeout on input.

Speech recognition differs from touch-tone input in that the application must start the recognizer each time input is to be collected from the caller. Also, after receiving IRE_INPUT_DONE, the recognizer is automatically turned off; the recognition request has been completed.

Speech recognition works closely with echo cancellation to support *barge-in* or recognition during prompting. Indeed, recognition during prompting is not possible unless echo cancellation is used.

This section first introduces the functions, parameters and header files used with speech recognition. Then, the chantest application shows the use of speech recognition with echo cancellation to support barge-in. This modified version of chantest stills supports input via touch tones and automatically turns off the recognizer when touch-tone data is received.

Refer to *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228, for additional information about speech recognition. Refer to *Intuity CONVERSANT VIS V5.0 Software Installation*, 585-310-151, for information on the software packages that must be installed support speech recognition (WholeWord and FlexWord).

Recognition Functions

The following functions are provided to support recognition:

- *irStartRecog(3IRAPI)* — Start speech recognition
- *irStopRecog(3IRAPI)* — Stop speech recognition
- *irCheckRecog(3IRAPI)* — Check the on/off status of speech recognition
- *irStartRecogTimer(3IRAPI)* — Start the speech recognition timer
- *irStartEcho(3IRAPI)* — Start echo cancellation
- *irStopEcho(3IRAPI)* — Stop echo cancellation
- *irCheckEcho(3IRAPI)* — Check the on/off status of echo cancellation

The input queue functions are also used to get and flush recognition data from the input queue. These include *irGetInput(3IRAPI)* and *irFlushInput(3IRAPI)* respectively. Refer to Appendix A, ‘‘Reference Material,’’ for additional information on these functions.

Recognition Parameters

The following parameters are used to control the behavior of the recognition functions:

- IRP_RECOG_TYPE — Indicates which type of recognition to use (IRD_WHOLE_WORD or IRD_FLEX_WORD)
- IRP_RECOG_GRAMMAR — Depends on IRP_RECOG_TYPE, valid values are found with the grammar header files associated with the recognition type
- IRP_RECOG_PRETIME — Specifies a timeout to wait for caller input, measured in milliseconds
- IRP_ECHOCAN_TYPE — Currently only one supported type IRD_SP_ECHO

NOTE:
It is best to leave the IRP_ECHOCAN_TYPE parameter alone. Changing it renders the echo canceler inoperable.

All recognition parameters should be set before calling any recognition functions and retain those settings until the functions complete.

WholeWord and FlexWord work similar from the perspective of the IRAPI. Once the recognition type and grammar is set, the function calls proceed in similar ways. The primary difference between the two types of speech recognition is that barge-in is not supported with FlexWord. However, the IRAPI does *not* prevent you from attempting to use the echo canceler or recognize during prompting with FlexWord; however, this results in premature responses from the recognizer with nonsense values. A similar situation arises when using the WholeWord recognizer during prompting without echo cancellation.

Grammar Header Files

WholeWord recognition grammars numbers are defined in header files. Each WholeWord recognition language has a specific header file named according to the following template:

/att/asr/grammar_hs/*{XX}*.gram.h

where *{XX}* is the country specific designation. For US English, *{XX}* is US. Applications using US English WholeWord recognition would include the grammar header file into their programs as follows:

```
#include "/att/asr/grammar_hs/US.gram.h"
```

All FlexWord recognitions use grammars defined in the **/att/asr/grammar_hs/sw_grammar.h** file. The grammars in this file are defined to have the 12th bit set. This flag, meaningful to TAS applications only, indicates to TSM that FlexWord is to be used. It should be removed before setting the grammar. The actual FlexWord grammar number is the defined value with the 12th bit reset. Therefore, IRAPI applications using FlexWord recognition must reset the 12th bit before starting the FlexWord recognizer. This is easily accomplished when setting the IRP_RECOG_GRAMMAR as follows:

```
irSetParam(cid, IRD_RECOG_GRAMMAR, WL_0 -  2048);
```

Recognition Events

Speech recognition uses only the IRE_INPUT_DONE event for the following reasons:

1. Recognition input is delivered from the voice cards atomically.
2. When recognition completes, it is done; there is no more input from the recognizer until it is restarted.

NOTE:
The IRE_INPUT event is never used.

This differs from touch-tone input where touch tones arrive from the voice cards one at a time and touch tones are continuously being recognized by those cards.

With touch-tone input, the IRE_INPUT_DONE event is generated when conditions on the input queue are met or the touch-tone timer expires. With speech recognition input, IRE_INPUT_DONE is generated regardless of the conditions on the input queue. The input length or delimiters settings do not effect the generation of the IRE_INPUT_DONE event in this case.

Echo Cancellation

Echo cancellation provides better speech recognition accuracy when attempting to recognize while prompting. Echo cancellation removes the echo produced while the prompt is being played from the caller input. Without echo cancellation, the recognizer performs poorly as it cannot distinguish the caller's input from the prompt's echo.

Echo cancellation uses an adaptive algorithm; therefore, the longer it is on, the better job it does. Applications requiring echo cancellation should turn on echo cancellation once the call is answered and leave it on until recognition during prompting is no longer required by the application.

In order to use echo cancellation, all voice output must be played on a SP circuit card. This is only relevant for T/R channels set to *talk* (that is, voice output is played on the T/R circuit card rather than an SP circuit card). If any such channels exist, the IRAPI automatically switches the channels to *tdm* (meaning voice output is played on an SP circuit card). Talk and tdm correspond to IRP_VOICE_TYPE settings of IRD_TALK_VOICE and IRD_SP_VOICE respectively. When using echo cancellation, the VOICE service must be assigned to an SP circuit card in the system. Refer to Chapter 3, ‘‘Configuration Management,’’ of *Intuity CONVERSANT VIS V5.0 Operations*, 585-310-550, to assign service to an SP circuit card.

Since echo cancellation performance improves over time, it remains on across *irExec(3IRAPI)* boundaries. *irExec(3IRAPI)*’d applications can use *irCheckEcho(3IRAPI)* to determine the status of the echo canceler.

Echo cancellation is automatically stopped when *irDeinit(3IRAPI)* is called.

In general, applications that use echo cancellation consume more SP resources than recognition applications that do not. Reasons include:

- The echo canceler must remain on, possibly for the entire application.
- The recognizer remains on for both the prompt and possibly for some time after the prompt completes.

Chantest Using Speech Recognition

This section shows how chantest uses speech recognition to get caller input. This modified chantest application allows recognition during prompting, thereby requiring echo cancellation. The caller enters touch tones to indicate that input will be touch tone rather than speech. After receiving the touch-tone input, the program turns off the recognizer.

chantest.c must be include the asr grammar header file.

```
#include "/att/asr/grammar_hs/US.gram.h"
```

setTTParams() must include setting of the recognition parameters.

```
int setTTParmas(channel_id cid)
{
    .  .  .
      irSetParam(cid, IRP_TT_INTERTIME, 5000) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_PRETIME, 5000) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_TYPE, IRD_WHOLE_WORD) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_GRAMMAR, US_4dig) == IRR_FAIL ||
      irSetParamStr(cid, IRP_INPUT_DELIM1, "#") == IRR_FAIL ) {
        return(-1);
        .  .  .
```

IRP_RECOG_GRAMMAR is set to US_4dig as defined in the header file included above. This grammar indicates that we are expecting to recognize exactly 4 spoken digits.

In general, the longer the echo canceler runs, the better job it does at canceling the echo from the output voice. *chantest* starts echo cancellation once the call is answered, allowing the echo canceler to run as long as possible before the caller is prompted. Starting the echo canceler is done asynchronously, the IRE_ECHO_START event is reported when starting the echo canceler completes. If successful, the chantest application continues by calling *startChanTst()*.

```
case IRE_ANSWER_DONE:
  if (irStartEcho(cid, 0) == IRR_FAIL) {
      cleanup("Error in irStartEcho", cid);
      break;
  }
         break;

case IRE_ECHO_START:
  if ( ev.event_mod1 == IREM_ERROR ) {
      cleanup("IRE_ECHO_START reports IREM_ERROR", cid);
      break;
  }
  startChanTst(cid);
  break;
```

Since chantest supports recognition during prompting, the recognizer must be started before the prompt is played. Prompts are queued and played from the *reprompt()* and *playInstr()* functions. Both of these functions are modified as follows:

```
void reprompt(channel_id cid)
{
    .  .  .

    if ( irStartRecog(cid,0) == IRR_FAIL ) {
        cleanup("irStartRecog Error", cid);
        return;
    }
    if ( irEnd(cid, 0, 0) == IRR_FAIL ) {

        .  .  .
}

void playInstr(channel_id cid)
{
    .  .  .

    if ( irStartRecog(cid,0) == IRR_FAIL ) {
        cleanup("irStartRecog Error", cid);
        return;
    }
    if ( irEnd(cid, 0, 0) == IRR_FAIL ) {

        .  .  .
}
```

Starting the recognizer before calling *irEnd()* allows the application to ensure that the recognizer is on before the prompt is played. If an attempt to start the recognizer is made after the call to *irEnd()* and the recognizer failed to start, the application prompts the user for input that it is not prepared to receive.

If the prompt completes before the recognition completes, the recognition timer is started. The call to *irStartRecogTimer(3IRAPI)* tells the recognizer that the prompt is complete and the recognition timeout, specified through the IRP_RECOG_PRETIME, now takes effect. From within the main while loop, the IRE_PLAY_DONE event for prompting is handled as follows:

```
. . .

case IRE_PLAY_DONE:
  switch(Chl[chan].PlayDone) {
    case REPROMPT_WHEN_DONE:
      reprompt(cid);
      break;
    case START_TIMER_WHEN_DONE:
      if (irCheckRecog(cid) == IRD_ON &&
        irStartRecogTimer(cid) == IRR_FAIL) {
          cleanup("irStartRecogTimer Failed", cid);
      }
      break;

      .  .  .
```

As stated earlier, if the caller enters a touch tone, the application must switch from using recognition to using touch-tone input. Again from within the main while loop, receiving the IRE_INPUT event that only occurs for touch-tone input causes chantest to stop recognition (if on) and start the touch-tone timer.

```
IRE_INPUT:
  if ( irCheckRecog(cid) == IRR_ON ) {
      (void) irStopRecog(cid);
      if ( irStartTTTimer(cid) == IRR_FAIL ) {
          cleanup("Can't start TT Timer", cid);
      }
  }
  break;
```

The IRE_INPUT_DONE event occurs when input has been received from the recognizer. The *input_done()* function now must handle recognition input while still supporting touch-tone input.

```
void input_done(channel_id cid, ir_event_t *evPtr)
{
    switch(evPtr->event_mod1) {
      case IREM_INPUT_LENGTH:
      case IREM_INPUT_DELIM:
      case IREM_RECOG:
        Chl[chan].RetryCount = 0;
        play_tt(cid);
        break;
      case IREM_TT_PRE:
      case IREM_TT_INTER:
      case IREM_RECOG_PRE:
        if(Chl[chan].RetryCount++ >= 3) {
```

If IRE_INPUT_DONE reports input, the input is played back to the user via *play_tt()*. Otherwise, the retry count is incremented and the script reprompts or quits as was done in the original chantest.

Resource Management

Resources are processing elements used to provide some voice, telephony, or call processing service. Voice system resources fall under two categories:

- Static resources

  These resources are always bound to a particular channel. These include touch-tone detectors, dialers, and play and record resources on T/R channels set to *talk.* These resources simply are used as they are needed since they are always available.

- Dynamic resources

  These resources exist in a pool from which resources are allocated as needed. Dynamic resources exist on SP circuit cards. Examples of dynamic resources include voice play and record resources on SP circuit cards and recognition resources. Since there may be fewer dynamic resources of a particular type than channels wishing to use them at any given time, IRAPI applications must deal with dynamic resource allocation contention.

NOTE:
Application developers need not be concerned about static resources. This section looks only at dynamic resource allocation and its effects on program behavior and structure. From this point forward, all discussion of resources refers to dynamic resources.

The IRAPI allocates all resources required to start an activity when the function requesting the activity is called. Implicit resource allocation is when resources required to complete a function or activity are allocated when a function is called. The IRAPI also allows an application to reserve any resources it may need in advance through a call to *irReserveResource(3IRAPI)*. This is termed explicit resource allocation.

Implicit Resource Allocation

Resources required to complete a function or activity are allocated implicitly when the function is called. The following functions implicitly allocate resources:

- *irStartEcho(3IRAPI)* — Allocates echo cancellation resources and an echo cancellation timeslot
- *irStartRecog(3IRAPI)* — Allocates recognition resources
- *irEnd(3IRAPI)* — Allocates SP play or TTS resources depending on whether play or say requests have been queued
- *irRecord(3IRAPI)* — Allocates SP record resources
- *irCall(3IRAPI)* — Allocates CCA resources if IRP_OUTCALL_CCALEVEL is set to IRD_FULL_CCA
- *irStartCCA(3IRAPI)* — Allocates CCA resources if IRP_OUTCALL_CCALEVEL is set to IRD_FULL_CCA

Resources are allocated when the function is called and freed when the activity completes, as indicated through an event.

The IRP_RESOURCE_RETURNMODE parameter determines the behavior of the IRAPI when all resources are busy during resource allocation. The following are valid settings for IRP_RESOURCE_RETURNMODE and the result of resource allocation failure if all resources of the requested type are busy:

- IRD_IMMEDIATE — Return IRR_FAIL
- IRD_BLOCKFOREVER — Return IRR_PENDING and wait indefinitely for IRE_GRANT
- *N* (where *N* is a timeout in milliseconds) — Return IRR_PENDING and wait up to *N* msec for IRE_GRANT, IRE_DENY, or corresponding IRE_<activity>_DONE event with an IREM_DENY modifier. Refer to Appendix A, ‘‘Reference Material,’’ for a complete description of the possible events associated with any function returning IRR_PENDING.

When a function returns IRR_PENDING, the IRAPI generates the IRE_GRANT event when the resource is granted. At that point no action is required from the application. The IRE_GRANT simply informs the application that the activity is now proceeding. If IRP_RESOURCE_RETURNMODE was set to some positive number and resources were not granted within the timeout represented by that number, the IRE_DENY event or the function specific IRE_<activity>_DONE event with the IREM_DENY modifier is generated. At that point, an application might perform some error processing under the assumption that resources will never become available.

Application developers must decide which resource return mode works best for their application. One setting may not be appropriate for all resource allocations. IRP_RESOURCE_RETURNMODE may be modified before each function call if an application is willing to wait for some resources but not for others. The following describes possible settings of IRP_RESOURCE_RETURNMODE based on application needs:

- IRD_IMMEDIATE

  If resources are not available, the function call fails and the application proceeds with its normal error processing at that point. However, IRD_IMMEDIATE may force applications to fail requests that may have been blocked only for a short period of time. An application that only occasionally encounters resource contention may drop calls unnecessarily. A simple application might use this value.

- IRD_BLOCKFOREVER

  Using IRD_BLOCKFOREVER may leave callers waiting around too long rather than dropping callers too quickly (as with IRD_IMMEDIATE).

- A timeout value reasonable for the application

  Specifying a reasonable time value is perhaps the best method for an application. It allows an application to incur some delays under fairly heavy loads and to take alternate action under extreme loads. The application must be able to effectively deal with delayed resource denial.

Allocation of channels via *irInit(3IRAPI)* and *irInitGroup(3IRAPI)* follows the same resource allocation strategy. Setting the *return_mode* argument to these functions determines their behavior with respect to delayed channel allocation. With channel allocation, however, the IRE_CHAN_GRANT and IRE_CHAN_DENY events are used to indicate a channel grant and deny respectively.

Delayed Resource Allocation Example

The speech recognition version of chantest provides a good example for showing the behavior of implicit resource allocation for the following reasons:

- It uses play, recognition, and echo cancellation resources.
- The play and recognition resource allocations are interrelated. That is, the play cannot start until recognition starts, meaning that the IRE_GRANT event triggers the prompt.

To make chantest support delayed resource allocation, IRP_RESOURCE_RETURNMODE must be set. After the channel is successfully initialized, from within the IRE_EXEC case of the main while loop, the parameter is set as follows:

```
if ( irSetParam(cid, IRP_RESOURCE_RETURNMODE,
  DELAYED_GRANT_TIMEOUT) == IRR_FAIL ) {
    cleanup ("Error on irSetParam", cid);
    break;
}
```

DELAYED_GRANT_TIMEOUT is defined elsewhere as 10000 milliseconds:

```
#define DELAYED_GRANT_TIMEOUT        10000
```

If speech recognition cannot be started due to insufficient resource (return code of IRR_PENDING), the play is not started, Chantest waits for the IRE_GRANT event before play is started. Recognition is started in the functions *playInstr()* and *reprompt()*. Both functions are updated as follows.

```
void playInstr(channel_id cid)
{
    . . .

    int ret;

    . . .

    ret =  irStartRecog(cid,0);
    if ( ret == IRR_FAIL ) {
        cleanup("irStartRecog Error", cid);
        return;
    } else if ( ret == IRR_PENDING ) {
        return;
    }

    . . .

}
```

Finally, event handling code must be added for IRE_GRANT and IRE_DENY, and IRE_ECHO_START must deal with the IREM_DENY modifier.

```
while (irWCheck(&ev) != IRR_FAIL) {
    . . .

    switch (ev.event_id) {

      . . .

      case IRE_ECHO_START:
        if ( ev.event_mod1 == IREM_ERROR || ev.event_mod1 == IREM_DENY ) {
            cleanup("IRE_ECHO_START reports IREM_ERROR/IREM_DENY", cid);
            break;
        }
        . . .


      case IRE_GRANT:
        if ( irLibState(cid) == IRS_PLAY_QUEUED ) {
            /* Recognition resources were delayed, start play now
             */
            if ( irEnd(cid, 0, 0) == IRR_FAIL ) {
                cleanup("Error on irEnd", cid);
            }
         }
         break;

       case IRE_DENY:
         /* Recognition resources denied.  Note: echo cancellation and
          * play resource allocation is indicated through an
          * an IRE_ECHO_START and IRE_PLAY_DONE event with an IREM_DENY
          * event respectively.
          */
         cleanup("Resources for recognition denied.",cid);
         break;

       . . .
    }
}
```

If the IRE_GRANT is due to play or echo cancellation, it is ignored and the program continues as normal. If echo cancellation resources are denied, the program drops the call. The program ignores play failures for any reason and simply continues, most likely attempting to reprompt.

Explicit Resource Allocation

Explicit resource allocation allows applications to reserve all the resources they are going to need, on a per channel basis, before they actually use them. By allocating resources in advance, applications can guarantee that the resources are available immediately at the time they are needed. Unfortunately, the allocated resources not being used cannot be used by any other application. Resource allocation is done through *irReserveResource(3IRAPI)*. Resources are reserved according to capability and implementation. Capabilities, defined in *IrRESOURCES(4IRAPI)*, are essentially the services provided by the resources. Current capabilities include:

- IRC_CCA — Call classification
- IRC_ECHOCAN — Echo cancellation
- IRC_RECOG — Speech recognition
- IRC_PLAY — Voice play
- IRC_RECORD — Voice record
- IRC_TTS — TTS

For each capability there are one or more implementations. Currently supported implementations of each capability are listed in *IrRESOURCES(4IRAPI)*. Applications can reserve resources without knowing which implementation they need by requesting the resource with implementation IRD_INVALID. This causes *irReserveResource(3IRAPI)* to reserve the default resource implementation. When resolving the default resource, *irReserveResource(3IRAPI)* consults the parameter switch associated with that capability. These parameter switches and the capabilities with which they are associated are also listed in *IrRESOURCES(4IRAPI)*. For example, IRC_PLAY is associated with IRP_VOICE_TYPE. If *irReserveResource(3IRAPI)* is called with capability IRC_PLAY and the implementation is set to IRD_INVALID, the play implementation resources associated with the setting of IRP_VOICE_TYPE are reserved. If IRP_VOICE_TYPE is set to IRD_SP_VOICE, an SP play resource is allocated. If IRP_VOICE_TYPE is set to IRD_TALK_VOICE, no resources are allocated since IRD_TALK_VOICE implies a static play resource bound to the channel.

The following example shows how resources could be pre-allocated for the speech recognition chantest program. Explicitly reserved resources are freed via a call to *irFreeResource(3IRAPI)* or *irDeinit(3IRAPI)*. Explicit resource allocations survive across *irExec(3IRAPI)* boundaries.

```
main()
{
    . . .

    static ir_reserve_t resourceRequest[] = {
        { IRC_PLAY, IRD_SP_VOICE },
        { IRC_ECHOCAN, IRD_SP_ECHO },
        { IRC_RECOG, IRD_WHOLE_WORD },
        { IRC_NULL, IRD_INVALID },
    };

    . . .

    while (irWCheck(&ev) != IRR_FAIL) {
        . . .

        switch (ev.event_id) {

          case IRE_EXEC:
            . . .

            /* After successful channel initialization */

            if ( irReserveResource(cid, 0, resourceRequest, IRD_IMMEDIATE,
              NULL ) == IRR_FAIL ) {
                cleanup("Explicit Resource Allocation failure", cid);
                break;
            }

          . . .

        }
    }
```

When to Use Explicit Resource Allocation

Explicit resource allocation should be used sparingly if at all. Explicit resource allocation causes all resources a channel requests to be bound to that channel regardless of whether the channel is actually using them. Consider a application where TTS is used only to speak out the results of a database lookup. The application is run on a 48 channel system with two TTS SP circuit cards and 1 VOICE SP circuit card. All other speech is recorded speech. If this application reserved TTS resources when it started, at most 12 channels could run simultaneously. Furthermore, the TTS resources are highly underutilized since the TTS resource is only used to speak out the results of the database lookup. The remainder of the call hold time is spent playing prompts and collecting touch-tone input. Therefore, this application is actually best served by using delayed implicit resource allocation.

Explicit resource allocation is best suited for systems running a mix of applications. A voice messaging system makes a fine example. There are two types of callers in a messaging system: those leaving messages and those retrieving messages. Assume that those leaving messages are of higher priority

since they are leaving messages about purchases they want to make. These messages are recorded using the IRAPI voice record functions. To be sure to minimize record setup times, so as not to confuse the caller with long delays, such an application should request record resources up front, thereby allowing access to these resources, when they are needed, regardless of load. The application may also reserve play resources as well to insure the highest level of service quality. The retrieving application does not reserve resources up front since the users (the message transcribers) are more willing to put up with small delays when retrieving messages.

Explicit resource allocation allows for a system level implementation of quality of service.

Other Resource Management Functions

Resources available on the system may be found by calling *irQueryResource(3IRAPI)*. This function returns a list of resources matching the query including a list of all cards that support the resource or function.

irRestrictResource(3IRAPI) allows an application to restrict itself to a set of resources. Only processes running as root may change resource restrictions. Resource restrictions survive across *irExec(3IRAPI)* boundaries.

Resource Management Strategies

As discussed earlier, resources are allocated to channels, either explicitly or implicitly, to serve the needs of the program. The Resource Manager (RM) driver is used to control access of resources across all processes attempting to use them.

Resources are described to RM when the system initializes or when SP cards are put into service. The usage vector, described in the following section, describes the load the resource usage imposes on the SP card.

RM uses an algorithm whereby the least functional, lightest loaded SP circuit card is selected. For example, if a system contains 2 SPs, one supporting voice record and play and the other supporting voice record and play, recognition and echo cancellation, and play resources are requested, RM selects from the SP supporting only record and play first since it is least functional.

When a card is gracefully taken out of service by a maintenance process, all current allocations are maintained but no new allocations are allowed. This allows allocations to be released from a card until the card is idle, at which time it is released to the maintenance process.

If a maintenance process forcibly removes a card from service, any current activities on the card are terminated. Any activities terminated in this way receive an IRE_<activity>_DONE event with the modifiers of IREM_ERROR and IRER_RESOURCE_REMOVED. For each resource explicitly allocated by a channel on the affected card, an IRE_RESOURCE_REMOVED event is generated by the IRAPI, informing the application of the resource removal.

Using rmdb to Assess Resource Utilization

The command line utility, **rmdb(1)** can be used to assess resource utilization and channel status.

The *-C* option of **rmdb** dumps the contents of the channel table maintained by the RM driver. Use **rmdb -C48,51** from the command line to show a dump of channels 48 and 51. The ''Allocated List'' and ''Pending List'' columns are of particular importance to resource allocation. This example shows that channel 48 is pending on a play resource while channel 51 has a play resource allocated.

```
Channel table:
    48  :    ownDev 11    type 1   ownQ 119
             defOwn 35801 deferPid -1   pendingForce 0  profile ptr 0xd1406000
             work 0x00000000
             timerID 397267                                      nTimers 1
                when 730309(+809)  tag 0x00000001  fn rmTimeoutResource  (0)
             Allocated List:  empty
             Restricted List:  empty
             Pending List:
                SP_PLAY(0)tag 0x1   card -1   allocNo 0x0      next 0x0
             Change Own Pending List:  empty
    51  :    ownDev 10    type 1   ownQ 120
             defOwn 35801 deferPid -1   pendingForce 0 profile ptr 0xd1403000
             work 0x00000000
             timerID 397020                                      nTimers 0
             Allocated List:
                  SP_PLAY(2) tag 0xfeed card 4 allocNo 0x20000 next 0x0
             Restricted List:  empty
             Pending List:  empty
             Change Own Pending List:  empty
```

rmdb also can be used to check the resource utilization of SP circuit card. The following example uses **rmdb -c8,10** from the command line to show a dump of the card table maintained by the RM driver. Of interest here are the Saturated, Highwater, and Current usage vectors and the usage vectors for the packfile resources such as SP_WW_RECOG and SP_PLAY.

```
    8   :       MTC_INSERV      0               0x00000000
        3   :   /vs/pack/sp.pack.69     4        functions
                2   "SP_WW_RECOG    "   [416    0       1666    0     ]
                3   "SP_ECHOCAN     "   [208    0       1666    0     ]
                0   "SP_PLAY        "   [208    0       0       208   ]
                1   "SP_RECORD      "   [208    0       0       208   ]
                Saturated     : [10000  10000   20000   10000 ]
                High Water    : [3744   0       19992   624   ]
                Current       : [0      0       0       0     ]
                Allocation #  : 0x0        0x0        0x0        0x0
    10  :       MTC_INSERV      0               0x00000000
        0   :   /vs/pack/sp.pack.1      2        functions
                0   "SP_PLAY        "   [208    0       0       208   ]
                1   "SP_RECORD      "   [208    0       0       208   ]
                Saturated     : [10000  10000   0       10000 ]
                High Water    : [1040   0       0       1040  ]
                Current       : [0      0       0       0     ]
                Allocation #  : 0x0        0x0        0x0        0x0
```

NOTE:
In the following discussion on SP resource utilization, an SP refers to the SP and any CMP card associated to that SP.

SPs are complex resources made up of multiple hardware and software components. Utilization of SP resources by a function cannot be described with a single value. Usage vectors allow for the description of SP functional usage across multiple components. For the purposes of practical guidance in understanding the data provided with **rmdb -c**, think of the usage vector as abstract components of the SP. Each function uses some number of units from each of the four components. For example, SP_PLAY uses 208 units of components 1 and 4 and 0 units of components 2 and 3.

The Saturated vector shows how many units of each component are available on the SP circuit card. The High Water vector shows the highest level of utilization the card has had since it was put in service. The Current vector shows the current level of utilization.

Administrators may check this data periodically or when experiencing load related problems to assess the level of SP utilization and potential for delayed resource allocation.

Text-to-Speech

This section describes the IRAPI functions that support Text-to-Speech (TTS) play and control. TTS requires one or more SP circuit cards capable of running TTS and the installation of the Intuity CONVERSANT VIS 5.0 Text-to-Speech Package. Refer to the hardware installation book for your platform and the *Intuity CONVERSANT VIS V5.0 Software Installation*, 585-310-151, for additional information about the hardware and software requirements for the TTS package.

ASCII text may be queued for play using any of the text queuing functions described below. Play commences when the *irEnd(3IRAPI)* function is called. Voice and text may not be queued together. Any voice play requests that are queued must be played with *irEnd(3IRAPI)* and play must complete before text may be queued for play.

TTS Queuing

The following functions may be used to queue ASCII text for play:

- The *irSay(3IRAPI)* function is used to queue text from an open file descriptor. This file descriptor is obtained from a call to *open(2)*.

  NOTE:
  This is the standard UNIX system call *open(2)*, not the *irOpen(3IRAPI)* function used to obtain a voice file descriptor.

- The *irBSay(3IRAPI)* function is used to queue text from a buffer.
- The *irFSay(3IRAPI)* function is used to queue the entire contents of a specified file.

TTS Play and Control

The following functions are used to control playing of queued text:

- The *irEnd(3IRAPI)* function is used to start the play of queued text. This puts the IRAPI in the IRS_SAYING state [see *IrSTATES(4IRAPI)*]. The IRF_MORE flag may be used with this function to allow more text to be queued and played while the IRAPI is in the IRS_SAYING state. (This flag is not valid for voice play requests.) If the IRF_MORE flag is not used with *irEnd()*, the application must receive an IRE_SAY_DONE event to indicate play is complete before more text may be queued.
- The *irStop(3IRAPI)* function may be used to stop TTS activity on a channel before normal completion. Saying is stopped when the application receives an IRE_SAY_DONE event.

NOTE:
The *irPlayResume(3IRAPI)* and *irGetVCount(3IRAPI)* functions cannot be used for TTS.

Applications should be written to handle the possible denial or delay of TTS resource allocation when *irEnd()* is used. Depending on the value of the IRP_RESOURCE_RETURNMODE parameter [see *irPARAMETERS(4IRAPI)*], *irEnd()* may return IRR_FAIL or IRR_PENDING if the TTS resource is not immediately available.

Platform Management

This section discusses various functions provided as an interface to the IRAPI platform, the IRAPI timer feature, errors, tracing and logging.

Platform Interface

An IRAPI process must register with the system upon startup with the *irRegister(3IRAPI)* function. This function takes a unique process name string as an argument. The process name must not exceed IRD_MAX_APP_NAME characters in length (see **irDefines.h**). The process name must be the same name given with the *-p* option to the **defService(1IRAPI)** command. The *irRegister(3IRAPI)* function returns the UNIX message queue key of the calling process if successful. IRR_FAIL is returned if an error occurs.

A process may obtain the UNIX message queue key of another IRAPI process by calling **irGetQKey(3IRAPI)** with the process name of that process.

The number of channels configured in the system may be obtained with the **irNumChans(3IRAPI)** function. It returns the number of channels which exist that are of the type(s) specified in its argument. Two types are supported: IRD_REAL and IRD_VIRTUAL. These values may be logically ORed together to obtain a total consisting of more than one type of channel.

Channel Service States

The service state of a channel can be obtained with the **irServiceState(3IRAPI)** function. There are several possible service states. The two most common are IRD_INACTIVE (channel is "on hook") and IRD_ACTIVE (channel is "off hook").

Library States

The IRAPI library state for a channel may be obtained with the *irLibState(3IRAPI)* function. This function may be used to test for an uncompleted activity that an application is performing on the channel. For example, if voice play is in progress, the library state is IRS_PLAYING. If the application has done an *irAnswer(3IRAPI)* and has not yet received the IRE_ANSWER_DONE event, the library is in the IRS_ANSWERING state. The many possible states that the IRAPI library may be in are described in *IrSTATES(4IRAPI)*.

Communicating with Other Processes

There are three functions described in *irPostEvent(3IRAPI)* which may be used to send a message to another process. One of these functions should be used instead of the old *mesgsnd(3SPP)* routine. The choice of which function to use is only a matter of convenience for the programmer. If the receiving process is an IRAPI process, it receives the message as an IRE_EXTERNAL event through a call to *irCheck(3IRAPI)* or *irWCheck(3IRAPI)*. Otherwise, the receiving process gets an IPC message through the *mesgrcv(3SPP)* function (see Appendix B, ''Voice System C-Library Functions,'' of *Intuity CONVERSANT VIS V5.0 Application Development*, 585-310-227).

- The *irPostEvent()* function requires a pointer to a message buffer and the length of the buffer as arguments. The application developer must fill the *irWhoTo* (destination queue key) and the *irChan* (channel number) fields in the message buffer before calling the function.
- The *irPostEventC()* function sends a message to whatever process owns a given channel. This function requires the channel number as an argument in addition to the message buffer and length.
- The *irPostEventQ()* function sends a message with a message queue key. It requires the queue key as an argument in addition to the message buffer and length.

Timer Management

The IRAPI provides a timer management facility with the following *irTimer(3IRAPI)* functions:

- The *irStartTimer()* function starts a timer for a specific channel. If the timer expires before it is canceled, an IRE_CLOCK event is triggered for the channel. Multiple timers per channel may be started as long as they are given unique *tag* values.
- The *irCancelTimer()* function cancels a channel specific timer. The same *tag* value used to start the timer must be used to cancel it.
- The *irStartPTimer()* function starts a process timer for the process calling the function. If the timer expires before it is canceled, an IRE_CLOCK event with a channel ID of IRD_NULL is triggered. Multiple process level timers may be started as long as they are given unique *tag* values.

Timer intervals are in milliseconds but have a 10 millisecond granularity. Timers may be set to go off once or repeatedly at the specified interval.

Errors, Tracing and Logging

Error messages may be logged by IRAPI applications using the same logMsg interface routines provided for Data Interface Processes (DIPs) described in Chapter 5, ''Adding and Modifying System Messages,'' and Appendix B, ''Voice System C-Library Functions,'' of *Intuity CONVERSANT VIS V5.0 Application Development*, 585-310-227. The *irRegister(3IRAPI)* function calls *logInit()*, eliminating the need for IRAPI applications to use *logInit()*. The *db_init()*, *db_pr()*, and *db_put()* functions are still available and work for printing messages to the system trace, but IRAPI processes should use the newer *irTrace(3IRAPI)* functions instead. The older functions may not be supported in the future.

NOTE:
Not all error conditions that occur within the IRAPI library are logged. The application developer must decide whether to log certain errors based on the return value of the IRAPI functions and the value of the *irError* variable.

The system trace facility provides a means of printing messages to a display terminal on which the **trace(1IRAPI)** command is being executed. Trace messages may be selectively output using process, channel, area and level parameters. There are 16 user defined areas and levels and 16 areas and levels reserved for the system. The *irTrace(3IRAPI)* functions support a variety of tracing operations:

- *irTrace()* and *irQTrace()* — Channel level tracing. Messages printed with these functions appear in the system trace if the specified channel is being traced. *IrQTrace()* is a macro that executes more quickly than *irTrace()*, but does not allow a variable number of arguments.
- *irTraceP()* and *irQTraceP()* — Process level tracing. Messages printed with these functions appear in the system trace if the process printing them is being traced. *IrQTraceP()* is a macro that executes more quickly than *irTrace()*, but does not allow a variable number of arguments.
- *irTrace_Put()* — Backward compatibility to *db_put(3SPP)*. This function prints a message to the trace output unconditionally.
- *irTRACECHAN_CHK()* and *irTRACEPROC_CHK()* — check tracing. These macros may be used to check to see if tracing is on for a particular channel or process before executing a block of code.
- *irSetTraceChan()*, *irSetTraceQkey()*, *irSetTraceArea()*, *irSetTraceLevel()*, *irSetTraceLogMode()*, and *irSetTraceDateMode()* — Set system tracing parameters. These functions may be used to change current values of the channel, process (by specifying the message queue key), area, level, log mode and date mode parameters of an executing trace command. (No output appears if a trace command is not running.)

The IRAPI contains many convenience functions that may be used with error and trace messages to print the symbolic names of IRAPI values. These functions are described in *irErrorStr(3IRAPI)* and *irName(3IRAPI)*:

- *irErrorStr()* — Character string describing an error
- *irErrorName()* — Symbolic name of an error code
- *irPError()* — Print to *stderr* an error description and user text
- *irPName()* — Parameter name [see *IrPARAMETERS(4IRAPI)*]
- *irSName()* — Library state name [*IrSTATES(4IRAPI)*]
- *irEName()* — Event name [*IrEVENTS(4IRAPI)*]
- *irEMName()* — Event modifier name [*IrEVENTS(4IRAPI)*]
- *irAName()* — Algorithm name [*IrALGORITHMS(4IRAPI)*]
- *irSvcStName()* — Service state name [*IrDEFINES(4IRAPI)*]
- *irCName()* — Capability name [*IrRESOURCES(4IRAPI)*]
- *irPrintEvent()* — Formatted string of event structure [*IrEVENTS(4IRAPI)*]

Application Dispatch Interface

The IRAPI maintains the concept of a default owner for a channel. The default owner is the process that receives ownership of idle, in service, channels. The default owner is responsible for handling IRE_NEWCALL events on channels that it owns, determining what application should service the new call and using *irExec(3IRAPI)* to invoke the application on the channel. Normally the default owner is the AD process. However, another process may change the default owner by using *irChDefOwn(3IRAPI)*.

The AD interface allows two different applications to be assigned to a channel. The *startup* application is the application that is run when AD receives an IRE_NEWCALL event. The *standard* application is the application that is run when another IRAPI application invokes the AD process with *irExec(3IRAPI)* and AD receives the IRE_EXEC event. When an application gives up channel ownership with *irDeinit(3IRAPI)* the default owner is notified with the IRE_DEFOWN event. Normally the startup and standard applications are identical. In special cases where it is desirable to have one application gather additional information about an incoming call before the application that actually handles the call is invoked, it may be convenient to assign them as different applications. (See the **assign** command *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230, for more details.)

The following code fragment illustrates how the standard AD process handles the IRAPI events involved in dispatching applications:

```
while (1)
{
    irWCheck();
    index = IRD_AD_STARTUP;
    case (event_id)
    {
              IRE_EXEC:
                  index = IRD_AD_STANDARD;
                  irInit(cid);

              IRE_NEWCALL:
                  iraQueryADTables(cid, index, application)
                  if (found)
                                                    irExec(.....);

                  if (not found)
                                                    log error

                  break;
              IRE_DEFOWNER:
                  irInit(cid);
                  break;

              default:
                  break;
    }
}
```

The IRAPI provides several functions for access to the AD interface. These may be used to implement an alternative to the standard AD process if desired.

- Initialize/add/delete AD entry in the channel and/or ANI/DNIS tables:

```
int iraInitADTables (int numchans, int numdnisani)
int iraInitADChannel (int numchans)
int iraInitADDnisani (int numdnisani)
int iraAddADChannel (int channel, int disp_mode, const char *reg_file)
int iraAddADDnisani (const IRA_STR_RANGE * dnisrange,
            const IRA_STR_RANGE* anirange, const char * reg_file)
int iraRemoveADChannel(int chan, int mode)
int iraRemoveADDnisani(const IRA_STR_RANGE * dnisrange,
            const IRA_STR_RANGE * anirange)
```

- Look up applications

```
int iraQueryADTables(channel_id cid, int mode, AD_APPL * appl)
int iraQueryADDnisani(int channel, int mode,
            const char * dnisstring, const char *anistring, AD_APPL *appl)
```

- Read through tables

```
int iraReadADChannel(int chan, int mode, AD_APPL *p_appl)
int iraReadADDnisani(AD_DNISANI_ENTRY *p_ad_dnisani_entry)
int iraRewindADDnisani()
```

Application Management

4

What's in This Chapter

This chapter discusses the steps necessary to compile and install an Intuity Response Application Programming Interface (IRAPI) application on the Intuity CONVERSANT Voice Information System (VIS). The compile and install procedure uses the chantest.c application, discussed in Appendix B, "Sample Application," as an example.

This chapter also discusses the various tools that are available to an application developer when trying to debug an IRAPI application.

Compiling and Installing IRAPI Applications

The following procedure details the steps necessary to compile and install an IRAPI application.

1. Compile an IRAPI application.

   The following shows the options and the libraries needed to compile an IRAPI application. This example uses chantest.c as the application:

   **cc -I/att/include -L/vs/lib chantest.c -o chantest **
   -lirEXT -lirAPI -lspp -lTOOLS -llog -lprism

   All necessary libraries and header files are provided with the Intuity CONVERSANT VIS V5.0 Application Software package.

2. Install the executable file anywhere desired. It does not have to be installed in any particular place on the system.

3. Install the speech files with the UNIX **cp(1)** or **cpio(1)** commands in the location where they are referenced by the application. For example, the chantest.c application stores all its speech files in the **/speech/chantest** directory.

4. Define the service for the IRAPI application using the **defService** command. The following example shows how this is done for the chantest application:

   defService -n -p chantest -t P chantest

   The *-n* option specifies that default values should be used for all options not specified on the command line. The *-p* option specifies the process name to which the service belongs. (In this case the service and process names are identical.) The process name string must be identical to the name used by the process as an argument to the *irRegister(3IRAPI)* function. The *-t* option specifies that chantest is a *permanent* process. This process should be running then the voice system is started and continue running until the voice system is stopped. Refer to **defService** in the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230, for a description of other options.

5. Assign the service defined in Step 4 to a channel or dialed number in the same manner that TSM script services are assigned. Refer to the **assign service** command in the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230. The following example assigns the chantest service to channel 0:

   assign service chantest to chan 0

6. Run a permanent IRAPI application when the voice system is started. The recommended way to accomplish this is to add a file to the **/etc/conf/init.d** directory containing an **inittab(4)** entry for the IRAPI process.

Debugging IRAPI Applications

The following tools are available to an application developer when debugging an IRAPI application. These tools include both UNIX and Intuity CONVERSANT VIS tools. The Intuity CONVERSANT VIS tools are noted below with an asterisk. For more information on the Intuity CONVERSANT VIS tools, refer to the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230. For more information on the UNIX tools, refer to the *UNIX SVR4.2 Command Reference*.

- vtlmgr
- trace*
- logCat*
- debug
- rmdb*

vtlmgr

If better X-windowing tools are not available, it is often useful to use the UNIX **vtlmgr** command to have multiple terminal sessions available.

trace

The **trace(1IRAPI)** command can be used to get detailed information about the program flow for IRAPI applications and other Intuity CONVERSANT processes.

In general, tracing an application is one of the best ways to debug an application. It provides minimal additional overhead on the process being debugged. In very rare cases, tracing may change the behavior of the problem by changing the timing of events.

The **trace** command options can be used to significantly alter the verbosity of the output. If in doubt about how much information to collect, it is best to collect more than enough information into a trace output file that can be searched later (using **vi**, **grep**, **awk**, etc.). When a small amount of trace output is expected, it may be useful to send the output through the UNIX **tee(1)** command. The **tee** command displays the output on the screen and saves it to a file for viewing later. When a large amount of trace information is expected, redirect the output directly to a file. Directing a large amount of information to the screen using **tee** causes delays in writing to the terminal, and thus messages may be lost.

The following example of the **trace** command provides detailed tracing information about TSM using the **tee** command:

trace tsm date chan all area all level all | tee /tmp/trace.out

The following example of the **trace** command provides detailed tracing information about TSM to the **/tmp/trace.out** file.

trace tsm date chan all area all level all > /tmp/trace.out

The *date* option causes the trace output to include date and time stamps. This helps establish the time between events and helps reconcile the trace output with events and alarms displayed by the **logCat** command (described below).

As described in the Chapter 3, "IRAPI Run-Time Services," IRAPI applications can use trace functions to provide application-specific tracing messages. These application-specific trace messages can complement the trace messages generated by internal IRAPI functions.

The TRACE_BUFFER_SIZE described in *irAPI.rc(4IRAPI)* determines the number of trace records maintained internally. If a larger value is used for this parameter, you have less risk of losing trace records because you exceeded the buffer size.

logCat

The **logCat** command can be used to display alarms and other events that occur during program execution. These messages often provide the first warning about a program error or other problem. The log messages supplement the messages available from **trace**.

debug

The UNIX **debug(1)** command can be valuable in locating program errors in IRAPI or other C-programs. Among many other things, this program can be used to:

- Get a stack trace for a program that has core dumped
- Single step through programs to trace program flow
- Set breakpoints and examine memory
- Set watchpoints to determine when memory locations change

This debugger (provided with UnixWare) is more powerful and easier to use than **sdb** which precedes it. Refer to the *UNIX SVR4.2 Command Reference* and the *UNIX SVR4.2 Programming in Standard C* for detailed information on the use of this command.

The following caveats should be noted when using this command:

- When debugging IRAPI applications, breakpoints and tracing should be used with caution since they can cause timers to expire while the program is suspended. You may introduce new problems that interfere with reproducing the original problem.
- Single stepping may fail to stop at the next statement or function.
- It is easier to debug application code for which you have source files than the internal IRAPI library routines. Without access to the IRAPI source code, it may be difficult to investigate problems that involve IRAPI library problems.

rmdb

The **rmdb** command can be used to display internal tables maintained by the Resource Manager (RM) and to control the verbosity of RM trace messages. Refer to ‘‘Using rmdb to Assess Resource Utilization’’ in Chapter 3, ‘‘Run-Time Services,’’ and the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230, for additional information on **rmdb**.

Performance and System Tuning for IRAPI Applications

5

What's in This Chapter

This chapter describes the resource management performance issues for IRAPI applications. This chapter also provides a list of the Resource Manager (RM) driver tunable parameters for the system. These parameters control the RM driver's capacity and behavior.

Resource Management

Application developers who use dynamic resources should be aware of the following information:

- Resources are associated with channels. When an application allocates a resource, it is attached to the channel. The resource can only be applied to the given channel. The application cannot share a single resource over a number of channels that it owns. In general, applications should return resources to the resource manager as soon as they are done with them.
- Dynamic resources are described to RM when the system initializes. The IRAPI uses these descriptions to drive the dynamic resources. This description comes in three parts.
  - — Functions are individual capabilities of which a dynamic resource card is capable. For instance, play, record, and whole word recognition are examples of functions.
  - — Each function is identified by a name and has a ‘‘utilization vector’’ that describes the load that it places on a resource card.
  - — Each time a function is invoked, it consumes the specified amount of the resource on the card.
- Resource cards are complex. The loads presented by various functions to resources are also complex. The IRAPI includes a facility for resource cards and functions to describe themselves to the platform. These descriptions drive the use of the cards.
- Function and resource card usage descriptions are represented as vectors. In order for the IRAPI to allocate a function to a card, the function’s usage description added to the card’s current usage must not exceed the maximum or saturated usage.
- Functions are grouped together into packfiles. The packfile description includes the name of the packfile and the list of functions that the packfile implements.

  When a resource card is brought in service, it is assigned a saturated usage and a packfile is loaded onto the card. The packfile includes the programs that run on the resource card as well as the description of the functions. The saturated usage of the card can vary depending on the physical configuration. For example, the number of companion (CMPs) circuit cards that are associated with an signal process (SP) circuit card modifies its saturated usage. Users can examine the usage values of resource cards using the **rmdb** command (refer to the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230).
- In general, the IRAPI tends to assign work to cards that run the fewest number of functions. For example, if a system contains two cards – one capable of recognition, echo cancellation, speech record and speech play and the second capable of only speech record and speech play – the

IRAPI tends to assign all speech play and record functions to the second, less capable card. If the second card eventually reaches its saturated usage, then the IRAPI starts assigning speech play and record functions to the first card. In cases where multiple cards have the same number of functions, the IRAPI spreads the load evenly on those cards.

- The IRAPI includes facilities for managing contention for scarce resources. If there are more requests for work than can be accommodated with the current set of resource cards, applications are free to wait for definite or indefinite amounts of time for the resource to become available. If the resource becomes free within those time constraints, the IRAPI allocates the resource to the application and notifies the application via an IRE_GRANT message. If the resource request cannot be filled within the time period, the IRAPI notifies the application via an IRE_DENY message.

  NOTE:
  Users can also examine pending requests using the **rmdb** command. Refer to the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230, for additional information.

  Applications can also choose to not wait for a resource. In this case, if the request cannot be filled immediately, the IRAPI does not notify the application when the resource becomes free.

  If a resource card is removed from service gracefully (that is, the immediate argument is not used), the IRAPI does not assign any more work to it until the card is returned to service. This allows the system to gracefully remove resource requests from cards with pending remove requests.

The following example of the **rmdb** command shows:

- Usage vector per function utilization of an SP
- Saturated usage is vector at maximum utilization
- High water mark is high historical value (since last start_vs)
- Current is current usage value

```
8   :  MTC_INSERV          0             0x00000000
       3   :  /vs/pack/sp.pack.69 4        functions
       2   "SP_WW_RECOG    "      [416          0      1666   0     ]
       3   "SP_ECHOCAN     "      [208          0      1666   0     ]
       0   "SP_PLAY        "      [208          0      0      208   ]
       1   "SP_RECORD      "      [208          0      0      208   ]
       Saturated     :     [10000        10000  20000  10000 ]
       High Water    :     [3744         0      19992  624   ]
       Current       :     [0            0      0      0     ]
       Allocation #  :     0x0        0x0         0x0         0x0
10  :  MTC_INSERV          0             0x00000000
       0   :  /vs/pack/sp.pack.1  2        functions
       0   "SP_PLAY        "      [208          0      0      208   ]
       1   "SP_RECORD      "      [208          0      0      208   ]
       Saturated     :     [10000        10000  0      10000 ]
       High Water    :     [1040         0      0      1040  ]
       Current       :     [0            0      0      0     ]
       Allocation #  :     0x0        0x0         0x0         0x0
```

Disk Performance

One of the major performance bottlenecks in the Intuity CONVERSANT VIS system is input and output (I/O) to the hard disk. For high-channel count applications, the application should be structured so that it presents the disk I/O subsystem with a manageable load.

The filesystem type chosen by default for the talkfiles (**/home2/vfs** by default) is the 8K Veritas filesystem. One of the principle performance advantages to using this filesystem is that it uses very large block sizes.

NOTE:
This advantage is traded off against space efficiency: if a file uses any portion of a block, the entire block is allocated. For example, if there is a 9K file, 3 4K blocks are allocated for it. The remaining 3K bytes are ‘‘wasted.’’

If you do not use the default speech filesystem, you should be careful to make sure that the filesystem type is appropriate. Otherwise, disk throughput suffers.

A single disk is capable of a finite amount of throughput. You can measure the throughput on a disk with the **sar -d** command as shown in the following example:

```
bop13# sar -d 5 5

bop13 bop13 4.2 1.1.2 i386    12/13/94

16:13:55  device   %busy   avque   r+w/s  blks/s  avwait  avserv

16:14:00   dsk-0       8     0.1       9      69     1.4     9.5

16:14:05   dsk-0       3     0.3       2      34     4.0    14.0
```

As the load on the disk rises, the percent busy (%busy) figure rises. As the disk gets busier, the average time to service (avserv) a request increases. Eventually, the disk is presented with more work than it can do and disk read and write requests take a long time to complete. For applications that have high I/O requirements (all 96 channel configurations qualify here), a second disk is necessary and you must ensure that your application balances requests between disks. The following methods may be used to balance the load between disks:

- Put the database on one physical disk and the speech filesystem on another
- Link individual directories between disks
- Use RAID via hardware or software
  - — Software RAID solutions (like the Veritas Advanced File System) help with load balancing through features like partition striping. Striping is a RAID technique for storing contiguous virtual sectors on separate physical disks. If a filesystem is striped across two disks, the even sectors would be on one disk and the odd sectors would be on the other. See the Veritas Advanced File System documentation for more details.
  - — Hardware RAID solutions are available commercially. To the voice system, these look like a single SCSI disk. Internally, they can be organized to enhance throughput or reliability.
- Use Network File System (NFS) to distribute access to speech files.

  NOTE:
  Before you rely on NFS, you should make sure that you understand the performance and reliability implications.

  Performance via NFS is complicated because there will be multiple machines making requests of a NFS server. The variability of the load requested by all of the clients of a given file server may be high. In order to service all of the requests in a reasonable time, you must ensure that both the server and the network are sized to handle the load. This sizing is highly application dependent.

  If you build an application that uses a NFS server, you should make sure that you have a configuration that delivers the kind of reliability that is appropriate for your application. You should experiment with NFS to make sure that behaviors like server or network crashes will not affect you or your customer's perception of reliability.

RM Tunables

The RM tunables are listed below. The default values for each parameter are listed the following table and the size refers to the size of each element. For example, NCHANNELS is sized at 121 channels by default and each channel entry consumes 360 bytes. The amount of space devoted to the channel table inside RM is 43,560 bytes.

NOTE:
These parameters are set to support most configurations of Intuity CONVERSANT VIS V5.0. Ordinarily, it should not be necessary for these parameters to be tuned. However, tuning may be necessary in some particularly challenging configurations.

NCHANNELS

The NCHANNELS parameter specifies the maximum number of channels configured in the system. There must be a channel entry for every type of channel: real, virtual and NONEX. In addition, there must be at least one virtual channel configured in the system.

NTDM

The NTDM parameter is not user-tunable and is listed here only for completeness. Users should not modify this parameter.

NCARDS

The NCARDS parameter specifies the maximum number of network interface and resource cards configured in the system. Cards that are not controlled by the voice system, like the central processing unit (CPU), the Ethernet, the Token Ring, the remote maintenance board (RMB), etc., do not require entries in the card table. This parameter normally should not have to be tuned.

NFUNCTIONS

The NFUNCTIONS parameter specifies the maximum number of functions that run on resource cards. Functions are things like SP_PLAY, SP_RECORD, SP_WW_RECOG, etc. This parameter normally should not have to be tuned.

NPACKFILES

The NPACKFILES parameter specifies the maximum number of packfiles running on resource cards that can be configured into the system. Each packfile supports a number of functions. This parameter normally should not have to be tuned.

NDEVICES

The NDEVICES parameter specifies the maximum number of applications that can simultaneously use the IRAPI. Each IRAPI-based process consumes a entry in the RM device table. When a process closes the driver (by exiting), the entry is returned to a free pool. The device table should be big enough to support the maximum number of IRAPI processes that are concurrently running. When the system runs out of entries in the device table, the following error is printed to the console:

```
RM TUNING ERROR:  Out of cloned devices
```

Users can query the number of free devices by using the **rmdb -d** command. Any entry with a process ID (pid) of -1 is available.

NDYNSTRUCT

The NDYNSTRUCT parameter specifies the number of dynamic resources that are available to RM. RM uses dynamic resource structures to keep track of:

- Allocated resources
- Pending allocations for resources, channels, and channel groups
- Restricted lists

If RM runs out of these structures, the IRAPI function fails with system errors and the following message is printed to the console:

```
RM TUNING ERROR:  Out of dynamicResourceList structures
```

Users can query the number of free resources by using the **rmdb** command:

```
# rmdb -D
  rmDynFreeListHead        0xd14915dc
  Entries on free list    128
```

NCHANNELGROUPS

The NCHANNELGROUPS parameter specifies the number of channel groups configured in the system. This is not a user tunable parameter. Each equipment group uses one channel group structure. This parameter is sized to correspond to the number of equipment groups. Processes internal to the Intuity CONVERSANT depend on allocating an appropriate number of channel groups.

NOTE:
The number of equipment groups can *not* be changed simply by changing this parameter.

PROFILE_SIZE

This parameter controls the size of the call profile. The call profile is allocated when a channel first takes a call and is never deallocated. So, if a channel never runs an application, no call profile will be allocated for it, and no memory will be consumed by the profile. Functions internal to the IRAPI depend on an appropriate size for the call profile. This parameter is not user-tunable.

Summary of RM Tunables

The following table summarizes each of the RM tunable parameters, its default, and its size.

Table 5-1. Resource Manager Tunable Parameters

Parameter Description	Parameter Name	Default	Size
Number of channels	NCHANNELS	121	360
Number of TDM busses	NTDM	3	1028
Number of network interface, resource cards	NCARDS	15	96
Number of functions supported by all packs	NFUNCTIONS	15	88
Number of different packfiles that can be installed	NPACKFILES	15	100
Number of processes that can simultaneously open RM	NDEVICES	64	220
Number of dynamic resource structures	NDYNSTRUCT	128	20
Number of channel groups	NCHANNELGROUPS	32	20
Size of the channel's profile	PROFILE_SIZE	3000	1

Parameter Tuning Procedure

To change the value of a tunable parameter, execute the following command

/etc/conf/bin/idtune <***parameter***> <***value***>

where <*parameter* is the name of the parameter for which you want to change the value and <*value*> is the new value for the specified parameter.

Global Parameters

The IRAPI supports a number of global parameters. These parameters are read only and system wide in scope. They are set in the file **/vs/data/irAPI.rc**. Note that this file also contains TSM specific parameters which do not apply to the IRAPI at large. See *irAPI.rc(4IRAPI)* in Appendix A, ''Reference,'' for a description of the global parameters.

Applications can use *irGetGlobalParam(3IRAPI)* to get an integer-type global parameter and *irGetGlobalParamStr(3IRAPI)* to get a string-type global parameter.

Reference Material

A

What's in This Appendix

This appendix contains manual pages for the library functions for the IRAPI. The information about the functions is discussed throughout this book.

The functions are listed in alphabetical order. Each function is on a separate page, and for each function, the following is provided:

- Function name and syntax
- Purpose of the function
- Effects of using the function
- Examples of the function

Each of these manual pages provided in this section are available on-line. At the system prompt, enter **man -M /vs/man** <***function name or data format***> where <*function name or data format*> contains the name of the function or data format that you wish to access. These manual pages are also accessible via the UNIX fingertip librarian. Consult the UNIX documentation for additional information.

irIntro

Name

irIntro — Introduction to IRAPI library functions

Synopsis

#include <**irapi.h**>

Description

The Intuity Response Application Programming Interface (IRAPI) provides a standard development interface for voice-telephony applications. The IRAPI is a high-level, C-language interface to provide both voice-processing and telephony functions. The IRAPI's capabilities include: voice recording, voice storage, voice playback, telephone touch tone sending and receiving, telephony call progress, speech recognition, text-to-speech processing, resource management and time-slot management. The IRAPI system includes a general mechanism for starting applications in response to network events.

Library Parameters

The IRAPI is designed to make it easy to write applications, while at the same time providing a rich set of options. These two goals may be in conflict if the complexity of all the optional behavior of the library is embedded in the same C-function calls that are required for simple access. The philosophy that guides these functions is to make simple functions as simple as possible. Where options are required, they are set outside of the call to the function. Reasonable defaults are provided for all functions. Default values for all parameters are described in *IrPARAMETERS(4IRAPI)*. For example, applications typically will record all phrases with a single coding algorithm. Calls to *irRecord(3IRAPI)*, do not require that the application specify the recording algorithm each time the function is called. If the recording algorithm needs to be modified, functions may use *irSetParam(3IRAPI)* to modify the IRP_RECORD_ALGO parameter.

All parameters are modified and queried through *irSetParam(3IRAPI)* and *irGetParam(3IRAPI)* respectively. All parameter names are prefixed with IRP_ to indicated the the symbol represents a IRAPI parameter. Parameters may be channel specific, process specific or system wide.

Error Processing

Failure of a IRAPI function is communicated through the return value of the function. Possible return values are documented per function. All possible return values are defined in *IrRETURNS(4IRAPI).* All IRAPI return codes are prefixed with IRR_. Applications can get additional information by examining the contents of the global IRAPI variable *irError* immediately following a function failure. Furthermore, the global IRAPI variable *irSysError* may be examined when *irError* is set to IRER_SYSERROR. All possible error codes are defined in *IrERRORS(4IRAPI).* All IRAPI error codes are prefixed with IRER_.

As a debugging aid, IRAPI provides the function *irTrace(3IRAPI)* to provide channel and process tracing over a range of user defined levels and areas.

Multi-threaded, Non-blocking Support

The IRAPI supports multi-threaded voice applications serving multiple telephone channels. A single-threaded application is one that runs on a single voice channel. A multi-threaded application is a single process that controls several channels. Multi-threaded applications manage each thread of the application explicitly. Usually, this involves maintaining state information for each thread and explicitly tracking the state of the application on each channel. A single-threaded application does not necessarily extend to a multi-threaded application easily. No special facilities are provided with the IRAPI for single-threading; single-threading is considered a special case of multi-threading.

Most IRAPI functions that request voice and telephony services are asynchronous (that is, non-blocking) functions. These functions return immediately after initiating a service request rather than blocking until the service is accomplished. When a function returns, control is returned to the application so that it can perform other duties while the requested service is being carried out. These other duties might include servicing a separate telephone channel running as a different thread in the same process.

Control is passed between the IRAPI and the application. Applications pass control to the IRAPI via *irWait(3IRAPI)* so that the library can service requests from hardware devices in real time. The primary blocking function in the IRAPI is *irWait(3IRAPI).* The application must call *irWait(3IRAPI)* frequently (more than once a second) to insure correct operation of the library. The library uses the *irWait(3IRAPI)* call to perform its own processing. Failure to call *irWait(3IRAPI)* frequently enough may not allow the library to process real-time events. This could result in staggered speech, incorrect call processing, or other errors. Applications should not invoke system calls that might block indefinitely [*read(2), write(2), msgrcv(2), poll(2)*] and semaphore operations should be used carefully (in non-blocking modes or in situations where the amount of time an application is blocked is very small); *pause(2)* and *wait(2)* should not be used at all. C library calls that rely on system calls that block indefinitely should not be used (for example, e.g. sleep(3IRAPI)*).* The developer must ensure that the application makes no blocking system calls as the library cannot protect itself.

Applications that need to do timing can block using *irWait(3IRAPI)* and use a UNIX signal to break from *irWait(3IRAPI)* when time expires or use the IRAPI timing routines to generate events at the desired times. When a signal is generated or a new event arrives, *irWait(3IRAPI)* returns. For example, *alarm(2)* can be used to generate a SIGALRM after a specified period of time. When a signal is received, *irWait(3IRAPI)* returns immediately. The *irTimer(3IRAPI)* function sets a clock for timeouts.

The IRAPI passes control back to the application by generating an event. After being called by the application, *irWait(3IRAPI)* blocks until an event is generated. Events are generated by many conditions. Most importantly, events are generated when asynchronously requested services are completed. Examples of service-completions include completing voice playback or recording, sending touch-tone digits and the timeout of the IRAPI clock [see *IrEVENTS(4IRAPI)* for a complete list of events].

The function *irCheck(3IRAPI)* returns information about the most recent event on the *event queue*. Events are processed in the order received. *irCheck(3IRAPI)* provides the event type, description, tag, and channel. Typically, an application issues the *irWait(3IRAPI)* and the *irCheck(3IRAPI)* routines in immediate succession when not performing other functions. For convenience, *irWCheck(3IRAPI)* combines *irWait(3IRAPI)* and *irCheck(3IRAPI)* into a single call.

The IRAPI lets an application associate an event with the specific service request (that is, specific IRAPI library calls) by allowing applications to associate a number ‘‘tag’’ with specific service requests from the IRAPI. Applications pass a tag to the IRAPI when voice or telephony services are requested. The tag is included with the event information provided by *irCheck(3IRAPI)* after the request is completed. The application must manage the tags; they are not interpreted by the IRAPI. Applications may use tags to map events specifically to previous service requests. Applications with multiple outstanding requests use tags to determine with which service request an event is associated.

Event and Interrupt Management

The IRAPI treats events and interrupts separately. Many events and interrupts have the same name [see *IrEVENTS(4IRAPI)*]. An **event** is the notification the IRAPI gives to an application when some condition occurs. An **interrupt** is the termination of voice/telephony functions when some condition occurs. Events themselves do not terminate voice/telephony functions. The IRAPI supports many events and interrupts and lets the application enable and disable both. Applications enable events for specific channels with *irSetEvent(3IRAPI)*. The application receives the event associated with a condition only if it has been enabled. *irSetEvent(3IRAPI)* can also disable the specified event for the channel. If an event is disabled, the application does not receive it.

Applications can turn interrupts on or off by name for each channel using *irSetEvent(3IRAPI)*. The IRAPI interrupts terminate voice/telephony functions

only if they have been enabled. In addition to the IRAPI interrupts under the control of the application, some conditions cause voice functions to terminate automatically (such as reaching the end of a voice file). These conditions are non-maskable events.

The application must manage both events and interrupts. Even if an interrupt has been enabled, the application does not receive notification of its associated event unless it is enabled with *irSetEvent(3IRAPI)*. Applications control whether touch-tone input will talk off speech by enabling or disabling the touch-tone arrival event (IRE_INPUT) an interrupt. If the interrupt is enabled, IRE_INPUT abruptly ends speech functions. For example, if IRE_INPUT is set as an interrupt, but not enabled as an event, then it talks off speech, but the application is not otherwise notified of the touch-tone arrival. Talkoff of the voice functions play, say, and record is done below the library control level. This is done to maximize talkoff performance, but it causes ambiguity between the arrival of the IRE_<ACTIVITY>_DONE event and the IRE_INPUT event. Applications should wait for both events to ensure that the activity has stopped and input is available on the input queue.

Resource Management

The IRAPI provides applications with access to abstract capabilities and does not require the application to manage the hardware resources required to provide those capabilities. The IRAPI capabilities include voice record and playback, touch-tone send and receive, telephone call progress processing, speech recognition, text-to-speech and speech verification. The application initially can bind to some or all system resources with *irRestrictResource(3IRAPI)*. Thereafter, the application is restricted to using only those resources, and they are managed dynamically to provide capabilities. This allows applications to ensure that they will not use more than a given number of resources.

The IRAPI attempts to make resources available when the application calls functions that name actions [for example, *irPlay(3IRAPI)*, *irRecord(3IRAPI)*] and keeps them available for the required time. The IRAPI can switch the same resources to another channel when resources are required there. If the requested capability is available, the requesting call returns immediately and the service is subsequently provided. If the requested capability is not available, the service request either fails and returns immediately, or returns the channel identifier *(*cid*)* a pending return code and places the *cid* into a pending library state. IRP_RESOURCE_RETURNMODE specifies whether a request returns immediately or waits. If the application chooses to wait, it can set a maximum interval to wait for the resource, after which the request fails.

For most IRAPI applications, the channel is the only static resource. Channels are opened with *irInit(3IRAPI)* and remain fixed to an application until *irDeinit(3IRAPI)* is called. The application need not fix any other resources. To provide application management of multiple channels, IRAPI routines require a *cid*.

While the IRAPI provides dynamic resource management, it also supports reserving, freeing, and querying of resources. A application might reserve resources when it needs to guarantee that a specific resource is available at a later time. *irReserveResource(3IRAPI)* fixes a set of resources to an application. *irFreeResource(3IRAPI)* releases resources. Applications should use *irReserveResource(3IRAPI)* only when necessary; in some instances, it may cause general voice system performance to deteriorate by blocking attempts to get resources that would otherwise succeed. *irQueryResource(3IRAPI)* indicates whether a resource is reserved, dynamically available, or not available.

The library provides consistent to resource allocation failure be it explicit, through *irReserveResource(3IRAPI)*, or implicit, through any implicit resource allocating function such as *irPlay(3IRAPI)*. IRP_RESOURCE_RETURNMODE determines the behavior of any function that may implicitly allocate resources. The behavior of *irReserveResource(3IRAPI)* is determined by the value of the *mode* argument. The following settings determine the behavior of functions which cannot allocate resources due to all the conditions where all requested resources are busy.

IRD_IMMEDIATE	Causes the called function to return with a IRR_FAIL immediately
IRD_BLOCKFOREVER	Causes the called function to return IRR_PENDING. The applications should then wait on IRE_GRANT before continuing.
Any positive integer *N*	Causes the called function to return IRR_PENDING. The application should then wait for either IRE_GRANT or IRE_DENY. IRE_DENY occurs if the resource could not be granted in *N* milliseconds.

Library States

Library states are used by the library to maintain information about channel activities and to prevent applications from attempting illegal operation sequences. Library states are defined in *IrSTATES(4IRAPI)*, all library state names are prefixed with IRS_. Library states are channel specific. Library states fall into 4 categories; idle, active, queued and pending. The only idle library state, IRS_IDLE, indicates that the channel is inactive or waiting for input. The active library states, IRS_<*function*>ING, indicate that the channel is performing some function <*function*>. The queued library states, IRS_<*function*>_QUEUED, indicates that the channel is queuing request from some function. The pending states, IRS_<*function*>_PENDING, indicate that the channel has requested,

either explicitly or implicitly, some resource which cannot be immediately granted. The current library state of a channel may be determined through *irLibState(3IRAPI)*, but that is seldom necessary. By properly responding to the events as they occur, an application seldom needs to concern itself with checking the value of the library state.

Voice Playing and Recording

The primary voice object in the IRAPI is the voice file. Voice files are similar to UNIX files. Voice files can be written and read with *irRecord(3IRAPI)* and *irPlay(3IRAPI)*. To do so, voice files are referred to with a voice file descriptor (vfd) that is analogous to a standard UNIX file descriptor. Vfds are obtained with *irOpen(3IRAPI)* and discarded with *irClose(3IRAPI)*. Vfds can be converted to UNIX file descriptors with *irVfd2Fd(3IRAPI)*. Vfds are used in order to track the state of files through a sequence of plays, codes, and closes. They allow the library to maintain the position of a played or coded file even though the file may not be used until sometime in the future. Voice file permissions and creating mode are manipulated with the IRP_CREATE_MODE parameter.

For complete control over playback and recording, the application can move to an exact position within the voice file in terms of real byte offsets [with *irLSeek(3IRAPI)*]. After playback or recording, obtain the number of milliseconds played or recorded with *irGetVCount(3IRAPI)*. Positions and counts within voice files are typically referred to in millisecond units. The *irTime2Byte(3IRAPI)* function takes a developer-specified time and translates it to a byte position within the voice file that is appropriate to the voice-coding algorithm being used. Conversely, *irByte2Time(3IRAPI)* translates byte position to time.

The *irFRecord(3IRAPI)* and *irFPlay(3IRAPI)* functions are provided for convenient manipulation of entire voice files. They are based on *irRecord(3IRAPI)* and *irPlay(3IRAPI)*. However, neither *irOpen(3IRAPI)*, *irClose(3IRAPI)*, nor a reference to a vfd is necessary with these functions. *irFRecord(3IRAPI)* and *irFPlay(3IRAPI)* process voice files as a whole, and do not permit positioning within a voice file.

In order to maintain compatibility with pre-IRAPI applications, the function *irTF2File(3IRAPI)* is provided to map talkfile and phrase numbers to files on the UNIX file system. Results from a call to this function may be used as arguments to the *irFRecord(3IRAPI)*, *irFPlay(3IRAPI)* or *irOpen(3IRAPI)* functions.

irRecord(3IRAPI), *irFRecord(3IRAPI)* and *irBRecord(3IRAPI)* are asynchronous routines that return immediately. When one of these routines is issued, the IRAPI enters the IRS_RECORDING library state. No other state modifying functions may be issued on that *cid* until a IRE_RECORD_DONE is generated.

irPlay(3IRAPI), *irBPlay(3IRAPI)* and *irFPlay(3IRAPI)* are asynchronous routines that return immediately. When one of these routines is issued, the IRAPI enters either the IRS_PLAY_QUEUED library state. Multiple voice files and buffers can

be queued and played sequentially in a seamless manner, such that the listener hears no *gaps* between them. To do so, *irPlay(3IRAPI)*, *irFPlay(3IRAPI)* or *irBPlay(3IRAPI)* must be called repeatedly to queue up multiple seamless plays. for a single channel. The end of the queue is marked by calling *irEnd(3IRAPI)*. Once *irEnd(3IRAPI)* is given, no further voice operations may be attempted on that *cid* until after the IRE_PLAY_DONE event is returned. If an application attempts another voice operation, the library returns an error. *irEnd(3IRAPI)* serves as an end-of-list indicator because play requests do not actually start until *irEnd(3IRAPI)* is executed. This allows the library make automatic play requests, thereby reducing the likelihood of gaps in output speech. IRE_PLAY_PROG is generated for the end of each individual voice file or buffer in the queue. When one play routine on the queue is interrupted, the entire sequence terminates. The application can determine where the termination occurred in the play sequence by counting the occurrence of IRE_PLAY_PROG.

Voice file recording has several enhancements. An answering machine tone precedes recording if IRP_RECORD_TONE is set to IRD_ON. Generally, *irRecord(3IRAPI)* and *irFRecord(3IRAPI)* record for *count* milliseconds, but can be set to terminate automatically after a period of silence specified by the IRP_RECORD_PRETIME and IRP_RECORD_INTERTIME parameters. The recording algorithm is specified with IRP_RECORD_ALGO.

In addition to voice file capabilities, the IRAPI provides routines for voice play and code to and from buffers in the process' space. *irBPlay(3IRAPI)* and *irBRecord(3IRAPI)* are used for voice play and code directly to and from the process' memory. Applications that use *irBPlay(3IRAPI)* and *irBRecord(3IRAPI)* must manage their own buffers.

Applications can use *irStop(3IRAPI)* to stop the library from playing or recording speech. The IRAPI stops the speech and return an IRE_PLAY_DONE or IRE_RECORD_DONE event to indicate that the sequence of plays or the record has been terminated.

The relative record and play gains can be set with the IRP_PLAY_GAIN and IRP_RECORD_GAIN parameters respectively.

irGetAlgorithm(3IRAPI) allows applications to find the speech coding algorithm of a voice object. *irConvertAlgorithm(3IRAPI)* allows applications to convert between algorithm types.

irPhReserve(3IRAPI) allows applications to reserve space in the filesystem for subsequent voice coding.

irPlayResume(3IRAPI) and *irRecordResume(3IRAPI)* allow applications to resume a play or record request from the point at which the activity was interrupted, plus or minus some offset. In fact, a prior record request may be resumed for playing and vice versa.

Applications access text-to-speech (TTS) utilities via the IRAPI. The function *irSay(3IRAPI)* speaks out a portion of a file, *irBSay(3IRAPI)* speaks out a text buffer, and *irFSay(3IRAPI)* speaks out an entire file. When one of these routines is issued, the IRAPI enters a IRS_SAY_QUEUED library state. Once the *irEnd(3IRAPI)* is given the IRS_SAYING state is entered, no further voice

operations may be attempted until after a IRE_SAY_DONE event is returned. If an application attempts another voice operation, the library returns an error.

Maintenance and Administration

The IRAPI provides a set of routines for rudimentary maintenance and administration. For administration, *irListResource(3IRAPI)* returns a list of all resources on the system including their busy status.

Telephony

The IRAPI provides direct support for tip/ring (T/R), T1 bit-oriented signaling, line-side T1 (LST1), and Primary Rate Interface (PRI) telephony, both for incoming and for outgoing calls. The library hides many of the details of the specific telephony type from the application. In cases where a telephony action is not supported for a given telephony type assigned to a channel (for example, flash on a PRI channel), the library generates errors.

The primary mechanism for outbound calling is *irCall(3IRAPI)*. This non-blocking routine handles dialing and causes event generation regarding call progress. *irDial(3IRAPI)* simply outputs a dial string. It does not otherwise analyze call progress. Both *irCall(3IRAPI)* and *irDial(3IRAPI)* accept a dial string that can include call-control symbols in addition to dual tone multi-frequency (DTMF) characters [see *IrDIALSTRINGS(4IRAPI)*].

Parameters are provided to change the behavior of outgoing calling. IRP_OUTCALL_DIALTYPE sets the type of dialing (for example, rotary, touch-tone). For channels supporting multiple levels of call classification, IRP_OUTCALL_CCALEVEL sets the level. IRP_FLASH_DURATION sets the length of the switch-hook flash performed by *irFlash(3IRAPI)*.

Call classification is implicit with *irCall(IRAPI)* however, call classification may be started at any time with *irStartCCA(3IRAPI)*. *irStopCCA(3IRAPI)* and *irCheckCCA(3IRAPI)* allow applications to stop or check the status of call classification.

When an incoming call is detected (by receiving a IRE_NEWCALL event), the application answers it with the *irAnswer(3IRAPI)*. Applications disconnect with *irDisconnect(3IRAPI)*. Through *irAnswer(3IRAPI)* and *irDisconnect(3IRAPI)*, applications manipulate the service state of the channel. The current service state of a channel may be found with *irServiceState(3IRAPI).* After a call is answered, touch tones are constantly being collected in an input queue. *irFlushInput(3IRAPI)* flushes the queue and *irGetInput(3IRAPI)* copies the contents of the queue to a buffer in the application.

irGetInput(3IRAPI) is commonly used to obtain a input string satisfying certain parameters, and the IRAPI provides several parameters to define input string characteristics. IRP_INPUT_LEN sets the number of characters defining a string. IRP_INPUT_DELIM1 and IRP_INPUT_DELIM2 specify the delimiters (1

or 2 characters) for a string. The number of milliseconds to wait for a string to begin [after *irStartTTTimer(3IRAPI)* starts the timer] is set with IRP_TT_PRETIME. The number of milliseconds to wait between digits is set with IRP_TT_INTERTIME. If any of these conditions is satisfied, the library generates a IRE_INPUT_DONE event, at which time the application typically would obtain the string through *irGetInput(3IRAPI)*. *irUngetInput(3IRAPI)* places a specified character back onto the queue.

The call profile contains all of the information available about an individual call. The amount and type of information depends on the telephony type and the particular call. Standard T/R offers essentially no call information. T1 offers dialed number identification service (DNIS). Integrated Services Digital Network (ISDN) PRI and Adjunct/Switch Application Interface (A/SAI) offer many more information elements, including Automatic Number Identification (ANI). Applications can get and set information elements of the call profile by using *irGetIE(3IRAPI)* and *irSetIE(3IRAPI)*.

TDM Bus Management

The IRAPI provides functions for managing the time division multiplexing (TDM) bus and network interface connections to the bus. These functions apply only to network interfaces that are connected to the same TDM bus (as in systems with multiple TDMs). Two parties can connect their TDM time-slots with *irHBridge(3IRAPI)*. An application can monitor an arbitrary channel with *irMonitor(3IRAPI)*. A monitor listens to all input and output on the specified channel. Applications can start and stop background music with *irStartBGPlay(3IRAPI)* and *irStopBGPlay(3IRAPI)*, respectively, and control the volume of the background music with the IRP_OUTPUT_BGGAIN parameter.

Language Support

The IRAPI includes capabilities to speak numbers and characters with proper inflection. Applications invoke *irSpeakNum(3IRAPI)* to speak numeric quantities and *irSpeakChar(3IRAPI)* to speak alphanumeric strings.

Application Control

The IRAPI supports three kinds of applications:

Transient Processes: These are standard UNIX Processes using the IRAPI. They come into existence when they are started by the Application Dispatch (AD) process or some other agent (inittab, the UNIX shell, etc.) and run until they complete.

One transient process may control several channels

simultaneously.

Scripted Applications, Run by an Application Engine:
(for example, TAS Programs, run by TSM). Developers generate TAS programs using script language or through Script Builder. These programs are executed by the TSM process. TAS programs can communicate with data interface processes (DIPs). Other types of applications can run under different application executives (for example, VoiceTek object files run by a custom application executive).

Permanent Processes: These processes start before a call arrives and continue to exist after the call disappears. After the process starts up and initializes itself, it *parks* and waits for another application to call *irExec(3IRAPI)* to notify it of a new call. Generally, this other application is AD. An example of such a process is TSM (a UNIX process) that uses the IRAPI to execute TAS programs.

UNIX processes implementing applications register with the library through *irRegister(3IRAPI)*. Once registered, applications may call *irWait(3IRAPI)* and begin taking calls. Alternatively, an application may attempt to gain ownership of channels, whether for outbound or inbound call processing via *irInit(3IRAPI)*.

The *irExec(3IRAPI)* family of functions allow one application to invoke another. An application of any given type can exec an application of any other type. The new application replaces the old application from the channel's point of view. If the application invoking *irExec(3IRAPI)* is a TAS script running under TSM [that is, it invokes *irExec(3IRAPI)* through the *exec(3TAS)* script instruction], the script terminates and the new application starts on the channel (TSM continues to run). If the application invoking *irExec(3IRAPI)* is a permanent process, control passes back to its starting point. The process itself does not terminate. If the application invoking *irExec(3IRAPI)* is a UNIX process, it transfers ownership for all of the resources it owns, including the channel, to the new process.

Note that *irExec(3IRAPI)* does not perform exactly the same function as its counterpart in UNIX. Unlike the UNIX *exec(2)*, that replaces the running version of the process with the image of another and starting the new process, *irExec(3IRAPI)*'s process image is not replaced by the new application. The application is exec'ed at the caller's, or channel's point of view. The UNIX process may continue to exist and to run, but the voice application that continues to interact with the user on that channel is the exec'ed application.

When processes use *irExec(3IRAPI)*, they can pass information to the invoked process through a buffer in the call profile. Call data records can also be preserved. The call profile is preserved across *irExec*. Therefore, *irExec*'d applications can augment, rather than replace, call data from previous

applications. The routines *irGetIE(3IRAPI)*, *irGetParam(3IRAPI)*, *irGetCDRecord(3IRAPI)*, *irGetIEStr(3IRAPI)*, *irGetParamStr(3IRAPI)*, *irSetIE(3IRAPI)*, *irSetParam(3IRAPI)*, *irSetCDRecord(3IRAPI)*, *irSetIEStr(3IRAPI)* and *irSetParamStr(3IRAPI)* query and modify the call profile.

Applications may communicate with other applications, either non-IRAPI style DIPs or IRAPI applications through *irPostEvent(3IRAPI)* that generates a IRE_EXTERNAL event on the channel or process for which the event was directed. *irPostEvent(3IRAPI)* provides interprocess communication and data passing. A process may post events directly to another process. *irGetQKey(3IRAPI)* provides the facility to learn the queue key of a process previously registered via *irRegister(3IRAPI)*.

The AD process is not an integral part of the IRAPI. AD is an application written in terms of the IRAPI. It starts applications based on call elements of the call profile. While channels whose application dispatch function is handled by AD are idle, their application ownership points to AD [that is, AD has invoked *irInit(3IRAPI)* for the channel]. Therefore, new call indications are routed to the AD process. AD maintains tables that associate application names, types, and locations with patterns of new call information. If a new call's profile matches, AD invokes *irExec(3IRAPI)* to start the application on the channel.

Speech Recognition

Speech recognition is controlled by use of the *irStartRecog(3IRAPI)* and *irStopRecog(3IRAPI)* function. *irCheckRecog(3IRAPI)* provides the status of the recognizer. Recognition input is provided on the same input queue as touch-tone input. The input parameters, IRP_INPUT_LEN IRP_INPUT_DELIM1, and IRP_INPUT_DELIM2, apply as well. Recognition provides its own timer so the function *irStartRecogTimer(3IRAPI)* is provided to control the timer. The parameter, IRP_RECOG_PRETIME, is used to provide an initial time out. There is no notion of interdigit timeout with speech recognition. The IRE_INPUT_DONE event is generated when recognition input arrives and the conditions implied in the above parameters have been met. The parameters IRP_RECOG_TYPE and IRP_RECOG_GRAMMAR determine the type and type specific grammar of recognition respectively. Unlike touch-tone input, the speech recognizer has to be restarted after each recognition input.

Applications use echo cancellation to improve recognizer accuracy when speech recognition is required during voice play. The IRAPI functions *irStartEcho(3IRAPI)*, *irStopEcho(3IRAPI) ,*and *irCheckEcho(3IRAPI)* provide the ability to start, stop and check echo cancellation status.

irAnswer

Name

irAnswer — Answer an incoming call

Synopsis

#include <**irapi.h**>

int irAnswer (channel_id *cid***);**

Description

The *irAnswer* function establishes the call between the calling party and the IRAPI application for the channel identifier (*cid*). For Tip/Ring (T/R) channels, this means taking the phone off hook. The channel service state is changed from IRD_INACTIVE or IRD_RINGING to IRD_ACTIVE.

Event

After the incoming call has been successfully answered, the IRE_ANSWER_DONE event is generated with the IREM_COMPLETE modifier. While the *cid* awaits this event, it is placed in the IRS_ANSWERING library state. For some telephony types, the IRE_ANSWER_DONE might be generated immediately without the need to temporarily go into the IRS_ANSWERING library state.

Return Value

IRR_OK is returned if the answer request is successfully initiated.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_BADSTATE if the channel is not in the IRS_IDLE library state

IRER_INVALID if the *cid* is invalid

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_NORESOURCES if the SP for the Primary Rate Interface (PRI) D channel is no longer available for answering the call

IRER_SERVICESTATE if the channel is not in the IRD_INACTIVE or IRD_RINGING service state (that is, the cid has already been answered)

See Also

irServiceState(3IRAPI)

irBGPlay

Name

irStartBGPlay, irStopBGPlay — Stop/start voice file playback in background

Synopsis

#include <**irapi.h**>

int irStartBGPlay (channel_id *cid***, int** *tag***, char *****voice_file***);**

int irStopBGPlay (channel_id *cid***, char *****voice_file***);**

Description

The *irStartBGPlay* function starts voice file (*voice_file*) playback in the background on a specified channel indicated by a channel identifier (*cid*).

The *cid* is obtained by calling *irInit(3IRAPI)*.

If *voice_file* is not playing already on some channel, it is started and its audio output is added to the normal voice response prompts on *cid*. Other channels may execute *irStartBGPlay* with the same *voice_file*. The audio then is added to those channels while still playing on *cid*. When *voice_file* is finished, it restarts play at the beginning and continues to play as long as at least one channel requires it.

The global parameter IRP_BACKGROUND_OVOL sets the background play output volume. The channel parameter IRP_OUTPUT_BGGAIN allows individual channels to apply a dB gain factor to the background volume. IRP_OUTPUT_BGGAIN must be set to the desired value before invoking *irStartBGPlay*. *irStartBGPlay* uses the value available at the time of playing background speech.

To drop background speech, invoke *irStopBGPlay*. System wide, background playback stops on a voice file when all channels requesting it have stopped it.

Each channel can have at most 7 output timeslots of which one is pre-assigned. So the channel can only add another 6 output timeslots.

Event

For *irStartBGPlay*, the IRE_BACKGROUND event occurs with *event_mod1* set to IREM_COMPLETE or IREM_ERROR to indicate success or failure, respectively, for background play start. If *event_mod1* is IREM_ERROR, *event_mod2* may be set to any error code specified below.

There is no event associated with *irStopBGPlay*.

Return Value

IRR_OK is returned if the request for background play is successfully initiated.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid

IRER_MAXCHAN_TIMESLOTS if the channel has reached the limit for output timeslots (7 maximum per channel)

IRER_SYSERROR if a driver call or system call failure occurs (see *errno* for additional information)

IRER_RESOURCEBUSY if the request to do background play cannot be granted due to insufficient resources (Note: no pending requests are possible with background play)

IRER_NORESOURCES if resources for background play do not exist

IRER_TSBUS if the *cid* is not on the TDM bus

See Also

IrPARAMETERS(4IRAPI), irPlay(3IRAPI)

irBusDisable

Name

irBusDisable — Disable a channel from talking or listening to the bus

Synopsis

#include <**irapi.h**>

int irBusDisable (channel_id *cid***);**

Description

The *irBusDisable* function disables a channel from talking or listening to the TDM (or other) bus. It is intended primarily for use by the maintenance processes at system startup and for use by *irDeinit* when a channel is released. It is not expected that application programs will need to use this function.

Event

No events are expected as a result of this function call.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

See Also

irInit(3IRAPI)

irByte2Time

Name

irByte2Time — Convert byte to time

Synopsis

#include <**irapi.h**>

long irByte2Time (int *algorithm***, long** *byte***);**

Description

The *irByte2Time* function returns the time equivalence in milliseconds of a specified number of *byte*s for a given *algorithm*. See *IrALGORITHMS(4IRAPI)* for the values of *algorithm*.

This function may be used by an application to convert a position in a voice file to time.

Event

No event results from the call to *irByte2Time*.

Return Value

Time is returned in milliseconds if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if an invalid value for *algorithm* is passed or if *byte* is negative.

Example

When a touch tone interrupts playback, a call to *irCheck(3IRAPI)* obtains an event structure that contains the event identifier (for example, IRE_INPUT). The application then calls *irGetVCount(3IRAPI)* which returns the byte-count from the point where the voice pointer was reset [see *irGetVCount(3IRAPI)*]. The *irByte2Time* function converts this byte count to time. The application can perform arithmetic on the time, for example to "rewind" five seconds, and convert this new time to a byte count with *irTime2Byte(3IRAPI)*.

Caveats

Silence compression is not supported. Text-to-Speech (TTS) byte to time conversion is not supported.

See Also

IrALGORITHMS(4IRAPI), irTime2Byte(3IRAPI), irGetVCount(3IRAPI)

irCCA

Name

irStartCCA, irStopCCA, irCheckCCA — Start/stop call classification analysis on a channel

Synopsis

#include <**irapi.h**>

int irStartCCA (channel_id *cid***, int** *tag***);**

int irStopCCA (channel_id *cid***);**

int irCheckCCA (channel_id *cid***);**

Description

When the IRAPI library parameter IRP_OUTCALL_CCALEVEL is set to IRD_FULL_CCA, the *irStartCCA* function starts Full Call Classification Analysis (Full CCA) on the specified channel *cid*. It is used by applications to start Full CCA independent of *irCall(3IRAPI)*. This function allows applications to use Full CCA, for instance, when doing a flash transfer via *irFlash* and dialing a string of digits via *irDial*. *irStartCCA* should be called before the *irDial* to ensure that the necessary SP resources are reserved and available to interpret the response to the dial.

When the IRAPI library parameter IRP_OUTCALL_CCALEVEL is set to IRD_BLIND_CCA or IRD_SIMPLE_CCA, *irStartCCA* does not need to be called since IRE_CALL_PROG events detected by the network interface card are generated (subject to event masking) regardless of whether *irStartCCA* has been executed. In other words, the IRAPI library acts as if CCA is always on for IRD_BLIND_CCA and IRD_SIMPLE_CCA.

irStartCCA does not need to be called when using *irCall* since *irCall* internally takes care of using *irStartCCA* when necessary.

The *tag* argument is a user-supplied number that associates the *irStartCCA* with a subsequent IRE_GRANT or IRE_DENY event. The *tag* included with IRE_CALL_PROG events is undefined (most likely a 0).

Various call progress events are generated in response to changing state of the call [see *IrEVENTS(4IRAPI)*]. The event is returned in an event structure (*ir_event*) that is accessed by *irCheck(3IRAPI)* or *irWCheck(3IRAPI)*.

IRAPI library parameters IRP_RESOURCE_RETURNMODE, IRP_OUTCALL_CCALEVEL, IRP_OUTCALL_ANSDET, and IRP_OUTCALL_MAXRINGS [see *IrPARAMETERS(4IRAPI)*] affect the operation of *irStartCCA*. IRP_RESOURCE_RETURNMODE indicates whether to wait for the necessary SP resources when Full CCA is being requested. IRP_OUTCALL_MAXRINGS specifies the number of rings to wait for before generating IRE_CALL_PROG event. IRP_OUTCALL_CCALEVEL specifies the CCA level (IRD_BLIND_CCA, IRD_SIMPLE_CCA, IRD_FULL_CCA) to be used for CCA. Simple CCA is done on the telephony hardware itself while Full CCA requires an SP dedicated to CCA. IRP_OUTCALL_ANSDET applies only to Full CCA and specifies whether answer supervision or speech energy detection should be used to determine that a call has been answered.

Values for these parameters can be retrieved using *irGetParam(3IRAPI)* while new values can be assigned to them using *irSetParam(3IRAPI)*. The default for IRP_OUTCALL_MAXRINGS is 5 and for IRP_OUTCALL_CCALEVEL is IRD_BLIND_CCA. The default for IRP_OUTCALL_ANSDET is IRD_DEFAULT which means that the most appropriate form of answer detection for the network interface is used. All the outcall related parameters must be set to the desired values before CCA starts.

Full CCA is turned off by default and is turned off automatically after a call classification has been made. Simple and blind CCA are turned on by default, and simple CCA is used when Full CCA is off. Running Full CCA only when required reduces load on the SP card(s) when IRD_FULL_CCA is used and therefore improves performance.

irStopCCA stops Full CCA on the *cid*. It leaves blind and simple CCA on, but returns IRR_OK so that programs can be written to have minimal differences when the CCA level is changed.

irCheckCCA returns the status of CCA (on or off).

Event

For *irStartCCA* only, IRE_GRANT and IRE_DENY indicate a return of IRR_PENDING returned if CCA resources are not available when requested. The *tag* provided with the IRE_GRANT or IRE_DENY event is the tag provided in the call to *irStartCCA*.

Regardless of the CCA level and whether Full CCA is on, CCA results are returned to an application via IRE_CALL_DONE or IRE_CALL_PROG event with different event modifiers. The *tag* provided for an IRE_CALL_PROG event is undefined (typically 0).

Return Value

For *irStartCCA* and *irStopCCA* IRR_OK is returned if the request is successful.

For *irStartCCA* only, IRR_PENDING is returned if CCA resources are not currently available and IRP_RESOURCE_RETURNMODE is not set to IRD_IMMEDIATE.

For *irCheckCCA* IRR_ON is returned if CCA is on and IRR_OFF is returned if CCA is off.

All functions return IRR_FAIL if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid or IRP_OUTCALL_CCALEVEL has an invalid value

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_BADSTATE for *irStartCCA* if the library state is not IRS_IDLE or IRS_CALLING

IRER_RESOURCEBUSY for *irStartCCA* if there are no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE

IRER_NORESOURCES if the CCA function has not been assigned to any in-service resource card

IRER_RESTRICTED if the channel is restricted from using the required resources

See Also

irSetParam(3IRAPI), irGetParam(3IRAPI), irCall(3IRAPI), IrPARAMETERS(4IRAPI), IrDEFINES(4IRAPI), IrEVENTS(4IRAPI)

irCDRecord

Name

irInitCDRecord, irGetCDRecord, irSetCDRecord — Initialize, get, or set an element of the call data record

Synopsis

#include <**irapi.h**>

int irInitCDRecord (channel_id *cid***);**

int irGetCDRecord (channel_id *cid***, int** *identifier***, int ****value***)**

int irSetCDRecord (channel_id *cid***, int** *identifier***, int** *value***)**

Description

These functions manage the call data that is stored in the call profile. The function *irInitCDRecord* initializes the call data record to zero. The function *irGetCDRecord* gets call data record number *identifier* and places it in the location pointed to by *value*. The function *irSetCDRecord* sets call data record number *identifier* to *value*.

Identifiers are simply integers ranging from 0 to IRD_MAX_CD that may have arbitrary meanings for end-user applications. The primary reason to use this call data feature is to allow call data to survive across *irExec* boundaries.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* or the *identifier* are invalid

IRER_SYSERROR if a driver call or system call failure occurs (see *errno* for additional information)

irCall

Name

irCall — Place a telephone call with call progress detection

Synopsis

#include <**irapi.h**>

int irCall (channel_id *cid***, int** *tag***, const char *** *dial_string***);**

Description

The *irCall* function places a telephone call to *dial_string* on the channel specified by *cid*. *irCall* automatically places *cid* in the IRD_ACTIVE channel service state.

The *tag* is a user-supplied number that associates the *irCall* with a subsequent event.

The *dial_string* is the telephone number of the call destination. Valid values for *dial_string* are listed in *IrDIALSTRINGS(4IRAPI)*.

The *irCall* function monitors call progress via Call Classification Analysis (CCA). Intermediate CCA results are returned to applications via a IRE_CALL_PROG event [*see IrEVENTS(4IRAPI)*]. Unlike *irDial(3IRAPI)*, *irCall* will automatically start and stop CCA as needed [*see irCCA(3IRAPI)*].

IRAPI library parameters IRP_RESOURCE_RETURNMODE, IRP_OUTCALL_DIALTYPE, IRP_OUTCALL_CCALEVEL, IRP_OUTCALL_ANSDET, and IRP_OUTCALL_MAXRINGS [see *IrPARAMETERS(4IRAPI)*] affect operation of *irCall*.
IRP_RESOURCE_RETURNMODE indicates what to do when resources are not immediately available. IRP_OUTCALL_DIALTYPE defines type of dialing to be used. Dialing types supported are DP (dial pulse), TT (touch tone) and MF (multiple frequency; for future use only). IRP_OUTCALL_CCALEVEL specifies the CCA level (IRD_BLIND_CCA, IRD_SIMPLE_CCA, IRD_FULL_CCA) to be used for CCA. Blind CCA does no call classification. Simple CCA is done on the telephony board itself while advanced CCA requires an SP card.
IRP_OUTCALL_ANSDET indicates how to detect that a call has been answered when Full CCA is being used. IRP_OUTCALL_MAXRINGS specifies number of rings to wait for an answer after dialing. If there is no answer within IRP_OUTCALL_MAXRINGS number of rings, an IRE_CALL_DONE event with an event modifier of IREM_NOANSWER is generated. Existing values of these parameters can be retrieved using *irGetParam(3IRAPI)* while a new value can be assigned to them using *irSetParam(3IRAPI)*. Default IRP_RESOURCE_RETURNMODE is IRD_IMMEDIATE,

IRP_OUTCALL_DIALTYPE is IRD_DIALTYPE_TT, IRP_OUTCALL_CCALEVEL is IRD_BLIND_CCA, IRP_OUTCALL_ANSDET is IRD_DEFAULT, and IRP_OUTCALL_MAXRINGS is 5.

WARNING: The parameters described above must be changed *before* calling *irCall* or *after* receiving the IRE_CALL_DONE event. Changing them while a call attempt is in progress can lead to unexpected results.

Calling *irStop(3IRAPI)* aborts (if possible) a call attempt that is in in the IRS_CALL_PENDING state after *irCall* has returned IRR_PENDING. Applications can also call *irStop* then *irDisconnect(3IRAPI)* to stop a call attempt in the IRS_CALLING state after *irCall* has returned IRR_OK (but before the IRE_CALL_DONE event is received). Depending on the telephony type, the *irStop* may not be able to terminate the call attempt before the IRE_CALL_DONE event is returned.

The *irDisconnect(3IRAPI)* function must be called to terminate a call after the IRE_CALL_DONE event has occurred. The application must call *irDisconnect* before attempting further calls even if the modifier to IRE_CALL_DONE indicates that the call was not completed (IREM_BUSY, etc.) or that an error occurred (IREM_ERROR). The application must also call *irDisconnect* before attempting further calls after receiving an IRE_DISCONNECT event. The IRAPI library automatically performs the *irDisconnect* when a channel is being de-initialized with *irDeinit(3IRAPI)*

Invocations of *irCall* cannot be intermixed on the queue with any other voice processing functions.

The channel will go to the IRS_CALL_PENDING library state if waiting for CCA or other resources. It will go to the IRS_CALLING library state while waiting for the IRE_CALL_DONE event.

Event

Intermediate call results are returned to an application via IRE_CALL_PROG event with event modifiers that provide more information about the status of the call. Final call results are returned to the application via the IRE_CALL_DONE event with event modifiers that report the disposition of the call. In some cases, additional IRE_CALL_PROG events may occur after the IRE_CALL_DONE event.

An IRE_CALL_DONE event with an IREM_GLARE event modifier may occur if there is already an incoming call on the *cid*. The recommended action is to call *irDeinit* to release the channel and to then allow the Application Dispatch process to pass the call to an application that will answer the incoming call.

Return Value

IRR_OK is returned if *irCall* is successfully initiated.

IRR_PENDING is returned if resources are currently not available and IRP_RESOURCE_RETURNMODE is *not* set to IRD_IMMEDIATE.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *dial_string* is too long or otherwise invalid or if the *cid* is invalid

IRER_SYSERROR if a system call failure occurs (see *irSysError* for additional information)

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_BADSTATE if the channel is not in the IRS_IDLE library state

IRER_RESOURCEBUSY if there are no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE

IRER_NORESOURCES if the CCA function has not been assigned to any in-service resource card

IRER_RESTRICTED if the channel is restricted from using the required resources

IRER_SERVICESTATE if the channel is not in the IRD_INACTIVE service state

See Also

irDial(3IRAPI), irSetParam(3IRAPI), IrEVENTS(4IRAPI)

irChDefOwn

Name

irChDefOwn — Change the default owner for a channel

Synopsis

#include <**irapi.h**>

ir_key_t irChDefOwn (int *chan***, ir_key_t** *msgqkey***);**

Description

irChDefOwn is used to set or change the default owner for a channel *chan*. The process that is to become the default owner is indicated by its message key *msgqkey*. The message key for the current process is obtained when calling *irRegister(3IRAPI)* and the message key for another process can be obtained by calling *irGetQKey(3IRAPI)*.

This function is primarily intended for use by maintenance and application dispatch processes, but can be used by applications that require an alternate application dispatch process. The process executing this command does not need to be the process identified by *msgqkey*.

The channel will be returned to the default owner whenever the last owner de-init's the channel by calling *irDeinit(3IRAPI)* or whenever the last owner exits via *exit(2)* or by a sudden exit (for example, dumping core).

This function will not change the current owner of the channel to the default owner; the new default owner must call *irInit(3IRAPI)* itself if it wants to assume ownership for the channel.

Event

No event is generated as a result of a call to *irChDefOwn*.

Return Value

If there was a prior default owner, the message key for the prior default owner is returned. If there never was a default owner for this channel, or if the default owner has exited, *irChDefOwn* returns a zero.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if *chan* is invalid.

IRER_SYSERROR on system or driver call failure (see *irSysError* for additional information).

See Also

irRegister(3IRAPI), *irGetQKey(3IRAPI)*

irChan

Name

irChanIS, irChanOOS — Place a channel in or out of service

Synopsis

#include <**irapi.h**>

int irChanIS (channel_id *cid***, int** *tag***);**

int irChanOOS (channel_id *cid***, int** *tag***);**

Description

The *irChanIS* function places a channel in-service. It is used by maintenance or related processes (such as MTC_PROXY) to place the channel in an idle or on-hook state as appropriate for the network interface. This function typically is called just before *irDeinit* when restoring a channel to service.

The *irChanOOS* function places a channel out-of-service. It is used by maintenance or related processes (such as MTC_PROXY) to place the channel in a busy-out or off-hook state as appropriate for the network interface. This function typically is called just after a channel has been granted via a call to *irInit* or *irForceInit*.

The *irChanOOS* function tells the remote switch or PBX that the CONVERSANT VIS will not accept incoming calls on this channel. Unfortunately, there is a brief period of time between the start of the *irInit* (or *irForceInit*) and the *irChanOOS* during which the remote switch could try to initiate an incoming call. If an incoming call occurs, the *irChanOOS* function waits for the completion of the busy-out or off-hook command. If an incoming call occurs, the IRAPI may internally attempt to clear the incoming call (that is, hangup on the caller) and again try to busy-out the channel. The automatic clearing of incoming calls is not a committed feature so there could be very rare cases where a caller gets connected to a disabled channel (the caller presumably hangs up after a few seconds even if the VIS does not).

Event

The *cid* is likely to temporarily be in the IRS_GOING_OOS library state while waiting for the completion of the *irChanOOS* request. The IRE_CHAN_OOS_DONE event is generated with the IREM_COMPLETE modifier when the request has been successfully completed.

The *cid* is likely to temporarily be in the IRS_GOING_IS library state while waiting for the completion of the *irChanIS* request. The IRE_CHAN_IS_DONE event is generated with the IREM_COMPLETE modifier when the request has been successfully completed.

Return Value

Both functions return IRR_OK when the request is successfully initiated.

Both functions return IRR_FAIL if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid

IRER_DRIVERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

See Also

irInit(3IRAPI)

irChan2Cid

Name

irChan2Cid — Convert a channel number to a channel identifier

Synopsis

#include <**irapi.h**>

channel_id irChan2Cid (int *channel_no***);**

Description

The *irChan2Cid* function returns a channel identifier (cid) associated with the *channel_no*. *cid*s primarily handle all channel based activities.

Event

No event results from the call to *irChan2Cid*.

Return Value

A valid *cid* is returned if the request is successful.

IRR_NULL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not owned by the process

See Also

irCid2Chan(3IRAPI)

irCheck

Name

irCheck — Get an event from the library event queue

Synopsis

#include <**irapi.h**>

int irCheck (ir_event_t ***_pir_event_);**

Description

irCheck is a non-blocking function used to access events. The events (IRE_*) and event structure (*ir_event_t*) are specified in *IrEVENTS(4IRAPI)*. *irCheck* must be called after *irWait(3IRAPI)* returns.

Whereas *irWait(3IRAPI)* blocks until an event occurs (and does not "return" the event), *irCheck* is used after a *irWait(3IRAPI)* to access the event.

irCheck returns information about only one event. If multiple events are queued, *irCheck* returns the first one.

If an event is queued, then a call to *irCheck* accesses the event and copies the event data into the caller allocated *ir_event_t* structure. The address of that structure is passed to the function as the *pir_event* argument. If an event is not queued, then a call to *irCheck* immediately returns IRE_NULL.

irCheck should be called until IRE_NULL is returned to clear the event queue.

Event

No event results from the call to *irCheck*.

Return Value

The *event_id* [see *IrEVENTS(4IRAPI)*] of the accessed event is returned if *irCheck* is successful.

IRE_NULL is returned if there are no events.

Caveat

The application must call *irWait(3IRAPI)* frequently to allow the library to handle the asynchronous returns for speech and telephony functions. If an application does not call *irWait(3IRAPI)* frequently enough, the library can not process asynchronous events in a timely manner and some functions may return errors or otherwise fail. For this reason, applications should not call any UNIX system calls which may block indefinitely.

irCheck is always used after a wait call returns an indication of a IRAPI event.

See Also

irWait(3IRAPI), IrEVENTS(4IRAPI)

irCid2Chan

Name

irCid2Chan — Convert a channel identifier to a channel number

Synopsis

#include <**irapi.h**>

int irCid2Chan (channel_id *cid***);**

Description

The *irCid2Chan* function returns a channel number associated with the channel identifier (*cid*). Channel numbers refer to hardware components and are useful when reporting errors.

Event

No event results from the call to *irCid2Chan*.

Return Value

A channel number is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if the *cid* is invalid

See Also

irChan2Cid(3IRAPI)

irClose

Name

irClose — Close a voice file

Synopsis

#include <**irapi.h**>

int irClose (vf_descriptor *vfd***);**

Description

The *irClose* function closes the voice file associated with a voice file descriptor (*vfd*). It is legal, but not required, to close a voice file after issuing an *irEnd(3IRAPI)* associated with an *irPlay(3IRAPI)* or *irRecord(3IRAPI)*.

Event

No event results from the call to *irClose*.

Return Value

IRR_OK is returned if *irClose* is successful. IRR_FAIL is returned if the *vfd* is not a valid voice file descriptor.

Error

irError is set to IRER_INVALID if the *vfd* is not a valid voice file descriptor.

See Also

irOpen(3IRAPI)

irConvertAlg

Name

irConvertAlgorithm, irFConvertAlgorithm, irBConvertAlgorithm — Convert speech to a new coding algorithm

Synopsis

#include <**irapi.h**>

int irConvertAlgorithm (int *tag***, vf_descriptor** *src_vfd***, vf_descriptor** *dest_vfd***, long** *count***, int** *algorithm***);**

int irFConvertAlgorithm (int *tag***, const char *****src_file***, const char *****dest_file***, int** *algorithm***);**

int irBConvertAlgorithm (int *tag***, const char *****src_buf***, const char *****dest_buf***, long** *count***, int** *algorithm***);**

Description

The *irConvertAlgorithm* function converts *count* **milliseconds** of speech in its current algorithm from the source voice file specified by the open voice file descriptor *src_vfd* to the specified algorithm (*algorithm*) and stores it in the destination voice file (*dest_vfd*).

irFConvertAlgorithm converts speech in its current algorithm from the source voice file *src_file* to the specified algorithm *algorithm* and stores it in the destination voice file (*dest_file*).

irBConvertAlgorithm converts *count* bytes of speech in its current algorithm from the source buffer *src_buf* to the specified algorithm (*algorithm*) and stores it in the destination buffer *dest_buf*.

Tag is a user supplied number that associates the conversion function call with a subsequent event. The event is returned in an event structure (*ir_event_struct*) that is accessed by *irCheck(3IRAPI)*.

Algorithm is a symbolically defined integer (IRA_*) that is defined in **irapi.h** [see *IrALGORITHMS(4IRAPI)*].

Event

The IRE_CONVERT_DONE event is returned with the level one modifier to indicate a successful or failed algorithm conversion. *tag* is included with the event.

Return Value

IRR_OK is returned if the algorithm conversion request is successfully initiated.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *src_vfd* or *dest_vdf* are invalid, if the *count* is negative, or if the *algorithm* is invalid

IRER_SYSERROR if a driver call or system call failure occurs (see *errno* for additional information)

Caveat

All algorithms (IRA_*) may not be supported by all IRAPI implementations. Use *irQueryResource(3IRAPI)* to check support for a specific algorithm.

Note that algorithm conversion is not associated with any *cid*.

See Also

irPlay(3IRAPI), irFPlay(3IRAPI), irBPlay(3IRAPI), IrALGORITHMS(4IRAPI)

irDeinit

Name

irDeinit — Deinitialize a channel

Synopsis

#include <**irapi.h**>

int irDeinit (channel_id *cid***);**

Description

The *irDeinit* function deinitializes a channel identified by the channel identifier *cid*.

Effects of *irDeinit* include:

1. The channel service state is set to IRD_INACTIVE
2. Any voice processing or telephony functions are stopped immediately
3. Reserved resources are freed
4. Any calls in progress are aborted

Outstanding events can not obtained after *irDeinit* is issued. Therefore applications should not deinitialize a channel before they are completely done with the channel.

NOTE: The *cid* must not be used after calling this function. The channel will no longer be controlled by this application.

Event

The event IRE_DEINIT_DONE is generated when the channel is released. This event does not apply to permanent applications; however, transient applications should wait for this event to occur prior to *exit(2)*ing. Waiting for IRE_DEINIT_DONE ensures that the channel is released cleanly.

Return Value

IRR_OK is returned if *irDeinit* is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

See Also

irInit(3IRAPI)

irDial

Name

irDial — Dial a dial string

Synopsis

#include <**irapi.h**>

int irDial (channel_id *cid***, int** *tag***, const char ***dial_string***);**

Description

The *irDial* function dials *dial_string* on the channel specified by *cid*. The *cid* must be in the IRD_ACTIVE service state before using *irDial*.

Tag is a user-supplied number that associates the *irDial* with a subsequent event.

The valid values for *dial_string* are listed in *IrDIALSTRINGS(4IRAPI)*.

IRAPI library parameter IRP_OUTCALL_DIALTYPE [see *IrPARAMETERS(4IRAPI)*] defines type of dialing to be used. The possible dialing types are IRD_DIALTYPE_DP (dial pulse), IRD_DIALTYPE_TT (touch tone) and IRD_DIALTYPE_MF (multiple frequency; for future use only). Not all dialing types are supported by all telephony types. The current dialing type can be retrieved using *irGetParam(3IRAPI)* while a new value can be assigned to it using *irSetParam(3IRAPI)*. For digital interfaces, the default IRP_OUTCALL_DIALTYPE is IRD_DIALTYPE_TT and cannot currently be changed. For TR and other analog interfaces, the Type of Signaling value from the Analog Interfaces screen (refer to "Switch Interface Administration" in *Intuity CONVERSANT VIS V5.0 Operations*, 585-310-550) is used as the default. This parameter must be set to the desired value before invoking *irDial*. *irDial* uses the value available at the time of placing the outcall.

The maximum length of the dial string depends on the telephony type. Currently all digital interfaces have a maximum dial string length of 15 characters. Applications that wish to pass more than 15 characters should use multiple *irDial* requests.

WARNING: The IRP_OUTCALL_DIALTYPE parameter must be changed before calling *irDial* or after receiving the IRE_DIAL_DONE event. Changing it while dialing is in progress can lead to unexpected results.

Event

The IRE_DIAL_DONE event is generated by *irDial* when dialing is completed.

Return Value

IRR_OK is returned if request is successfully initiated.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *dial_string* is too long or otherwise invalid or if the *cid* is invalid

IRER_BADSTATE if the *cid* is not in the IRS_IDLE library state

IRER_SERVICESTATE if the *cid* is not in the IRD_ACTIVE service state

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

Caveat

This function does not originate an outbound call on any telephony connections. Applications should use *irCall* to originate calls for generic channels. This function can be used to generate DTMF digits on any trunk type after a call has been set up. Applications using IRD_DIALTYPE_DP should be aware that this dial type may be inappropriate for active channels.

See Also

irCall(3IRAPI), irSetParam(3IRAPI), irGetParam(3IRAPI), IrPARAMETERS(4IRAPI), IrEVENTS(4IRAPI)

irDisconnect

Name

irDisconnect — Disconnect a call

Synopsis

#include <**irapi.h**>

int irDisconnect (channel_id *cid***, int** *tag***);**

Description

The *irDisconnect* function disconnects the call between the called/calling party and the IRAPI application for *cid*. For Tip/Ring (T/R) channels, this amounts to putting the phone onhook. The channel is moved from the IRD_ACTIVE to the IRD_INACTIVE service state. Depending on the telephony type, the service state may go through various intermediate states before getting to the IRD_INACTIVE service state.

A disconnect is performed automatically if *irDeinit(3IRAPI)* is called without first calling *irDisconnect*, but an IRE_DISCONNECT_DONE is generated only if *irDisconnect* is called directly by the application.

Event

After the call has been successfully disconnected, the IRE_DISCONNECT_DONE event is generated with the appropriate success/failure modifier. While the channel awaits this event, it temporarily is placed in the IRS_DISCONNECTING library state. Applications should wait for the IRE_DISCONNECT_DONE rather than checking for the IRS_DISCONNECTING library state.

Return Value

IRR_OK is returned if the dial request is successfully initiated.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_BADSTATE if the channel is not in the IRS_IDLE library state

IRER_INVALID if the *cid* is invalid

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_SERVICESTATE if the channel was not initially in the IRD_ACTIVE or IRD_RINGING service state

See Also

irDeinit(3IRAPI), *irServiceState(3IRAPI)*

irEcho

Name

irStartEcho, irStopEcho, irCheckEcho — Add/drop/check echo cancellation

Synopsis

#include <**irapi.h**>

int irStartEcho (channel_id *cid***, int** *tag***);**

int irStopEcho (channel_id *cid***, int** *tag***);**

int irCheckEcho (channel_id *cid***);**

Description

The *irStartEcho* and *irStopEcho* functions start and stop, respectively, echo cancellation on the specified channel *cid*.

Echo cancellation is used to improve speech recognition accuracy during speech playback. Echo cancellation is only improves accuracy if speech recognition is required during speech playback.

tag is used to correlate the function call with a subsequent event.

If successful, *irStartEcho* sets the *cid* to the IRS_ECHO_STARTING state. The *cid* is returned the IRS_IDLE state when the IRE_ECHO_START event is generated.

If successful, *irStopEcho* sets the *cid* to the IRS_ECHO_STOPPING state. The *cid* is returned to the IRS_IDLE state when the IRE_ECHO_STOP event is generated.

irCheckEcho returns the status of the echo cancellation function for *cid*.

Event

If resources are not available when *irStartEcho* is called and IRP_RESOURCE_RETURNMODE is *not* set to IRD_IMMEDIATE, an IRE_GRANT event occurs when resources become available. If resources do not become available within the time specified in IRP_RESOURCE_RETURNMODE, an IRE_ECHO_START event is generated with the IREM_DENY modifier.

Echo cancellation disposition cannot be assessed until the event IRE_ECHO_START is received. The *event_mod1* modifier of IRE_ECHO_START indicates whether the function was successful or an error occurred in starting the echo canceler or allocating resources. [See *IrEVENTS(4IRAPI)*.]

IRE_ECHO_STOP is generated after echo cancellation is successfully turned off via *irEchoStop*, if there is a spontaneous echo cancellation failure, or if the echo cancellation resource is forcibly taken out of service by a maintenance process. After IRE_ECHO_STOP is received, the application should discontinue attempts to use speech recognition during prompting. If echo cancellation was spontaneously stopped, applications may attempt to restart the echo canceler via *irStartEcho*.

Return Value

All functions return IRR_FAIL if an error occurs or if immediate resource allocation is not possible.

For *irStartEcho* and *irStopEcho*:

IRR_OK is returned if the request is successful.

For *irStartEcho* only:

IRR_PENDING is return if resources are not available and IRP_RESOURCE_RETURNMODE is not set to IRD_IMMEDIATE.

For *irCheckEcho*, the following additional return values are possible:

IRR_ON if the echo cancellation function is on

IRR_OFF if the echo cancellation function if off

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid

IRER_RESOURCEBUSY if there are no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE or there are no system timeslots available (regardless of the setting for IRP_RESOURCE_RETURNMODE)

IRER_NORESOURCES if there are no resources in existence

IRER_RESTRICTED if the channel is restricted from using the required resources

IRER_REDUNDANT if echo cancellation is already on in the case of *irStartEcho* or already off in the case of *irStopEcho*

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_SYSERROR if a system call failure occurs (see *irSysError* for additional information)

IRER_BADSTATE if the library channel state is not IRS_IDLE

See Also

irRecog(3IRAPI), IrDEFINES(4IRAPI)

irEnd

Name

irEnd — Mark the end of a list of play or say functions

Synopsis

#include <**irapi.h**>

int irEnd (channel_id *cid***, int** *flag***, int** *tag***);**

Description

The *irEnd* function marks the end of a list of one or more "play" [*irPlay(3IRAPI)*, *irBPlay(3IRAPI)*, *irFPlay(3IRAPI)*], or "say" [*irSay(3IRAPI)*, *irBSay(3IRAPI)*or *irFSay(3IRAPI)*] routines on a channel specified by *cid*.

The number of bytes played or recorded may be obtained by calling *irGetVCount(3IRAPI)*.

If *flag* is set to IRF_REMEMBER, the play activity is remembered. Use IRF_REMEMBER with *irPlayResume(3IRAPI)* or *irRecordResume(3IRAPI)* to mark resumable play requests which have been interrupted.

If *flag* is set to IRF_MORE, the say activities queued are started but the library allows more say requests to be queued. The library expects an *irEnd* to eventually be executed with flag set to 0 or the channel hangs in the IRS_SAY_QUEUED state. If applications take too long to queue text, callers may hear speech breaks.

IRF_MORE is ignored for queued play requests and IRF_REMEMBER is ignored for queued say requests.

Event

IRE_PLAY_DONE is generated when the play operation has completed. For a sequence of playbacks, a IRE_PLAY_PROG event is generated after each intermediate play function in the list except the last, and a IRE_PLAY_DONE event is generated when the last playback function in the list has finished playing.

IRE_SAY_DONE is generated when the last say operation has completed. A IRE_<*SAY_PROG*> type event is not implemented for say functions.

The IRE_PLAY_DONE or IRE_SAY_DONE event is generated with the IREM_DENY modifier if resources could not be allocated in the time specified by IRP_RESOURCE_RETURNMODE and *irEnd* returned IRR_PENDING. When resources are allocated after IRR_PENDING is returned, the IRE_GRANT event occurs.

Return Value

IRR_OK is returned if the request to play or say is successfully initiated.

IRR_FAIL is returned if an error occurs.

IRR_PENDING is returned if no resources are available for the play or say function.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_BADSTATE if *irEnd* is called without first calling one or more play or say functions, thereby putting the channel into the IRS_PLAY_QUEUED or IRS_SAY_QUEUED states

IRER_RESOURCEBUSY if there are no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE

IRER_NORESOURCES if there are no resources in existence

IRER_RESTRICTED if the channel is restricted from using the required resources

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_SYSERROR if a system call failure occurs (see *irSysError* for additional information)

See Also

irPlay(3IRAPI), *irSay(3IRAPI)*, *irStop(3IRAPI)*.

irErrorStr

Name

irErrorStr, irErrorName, irPError — IRAPI error messages

Synopsis

#include <**irapi.h**>

char *irErrorStr (int *error***);**

char *irErrorName (int *error***);**

void irPError (const char *****error*);**

Description

The *irErrorStr* function returns a pointer to a character string describing an IRAPI error (*error*). *irErrorName* returns a pointer to the symbolic name of the error defined in *error*. *error* must be one of the errors codes defined in *IrERRORS(4IRAPI)*.

irPError prints the user supplied string followed by the error string to standard error.

Event

No event results from the call to *irErrorStr*.

Return

For *irErrorStr* and *irErrorName*, the following returns are possible:

A pointer to the error message text is returned if the request is successful.

If *error* is not defined for IRAPI, a pointer to a undefined error string or symbolic name is returned.

Error

irError is not modified as a result of the call to *irErrorStr*, *irErrorName ,* or *irPError*.

See Also

IrERRORS(4IRAPI).

irEvent

Name

irSetEvent, irGetEvent, irInitEvents — Modify event control of library

Synopsis

#include <**irapi.h**>

int irSetEvent (channel_id *cid***, int** *event_id***, int** *action***);**

int irGetEvent (channel_id *cid***, int** *event_id***);**

int irInitEvents (channel_id *cid***);**

Description

The *irSetEvent* function allows the user to control the actions of the library when events occur on the channel specified by *cid*. Possible actions are the notification of the event or the notification plus interruption of playing, recording, saying, and/or dialing. Notification that the event has occurred is provided via *irCheck(3IRAPI)*. Interruption without notification is not valid.

Event_id is a valid event as specified in *IrEVENTS(4IRAPI).*

Action is an ORed list of possible library actions on the occurrence of the event defined as follows:

IRF_IGNORE	Ignore the event
IRF_NOTIFY	Notify on event
IRF_PLAYINTR	Interrupt playing
IRF_RECINTR	Interrupt recording
IRF_SAYINTR	Interrupt saying
IRF_CALLINTR	Interrupt out calling

If *action* is set to IRF_IGNORE, the event is disabled, that is, occurrence of the event is not reported by the library. Not all events may be disabled, see IrEVENTS(4IRAPI). If IRF_IGNORE is used with any other flags, IRF_IGNORE is ignored and the remaining flags are set. When *irSetEvent* is called, the current setting of the *event_id* action is overwritten with the new value specified in *action*.

irGetEvent returns the event action for *event_id*.

irInitEvents returns all events for *cid* to their default settings.

Default event assignments are described in IrEVENTS(4IRAPI).

Event

No event results from the call to *irSetEvent* or *irGetEvent*.

Return Value

The previous setting of the event is returned if *irSetEvent* is successful.

The current setting of the event is returned if *irGetEvent* is successful.

IRR_OK is returned by *irInitEvents* if it is successful.

IRR_FAIL is returned if an error occurs for any of these functions.

Error

irError is set to IRER_INVALID if the *cid*, *event_id*, or *action* are not valid or if *event_id* cannot be ignored and *action* is set to IRF_IGNORE.

Example

To set the IRE_INPUT event to interrupt play and notify the application when input occurs, the following code may be used:

```
channel_id cid;

...

irSetEvent(cid,IRE_INPUT,IRF_NOTIFY|IRF_PLAYINTR);
```

See Also

IrEVENTS(4IRAPI), IrERRORS(4IRAPI)

irExec

Name

irExecvp, irExecl, irExecp, irExecv, irExecle, irExecve, irExeclp — Transfer ownership of a channel to another Intuity Response application

Synopsis

#include <**irapi.h**>

int irExecp (channel_id *cid***, const char *** *service***, const char *** *name***, const void *** *pdata***, int** *psize***);**

int irExecl (channel_id *cid***, const char *** *service***, const void *** *pdata***, int** *psize***, const char *** *path***, const char *** *arg0***,**

int irExecv (channel_id *cid***, const char *** *service***, const void *** *pdata***, int** *psize***, const char *** *path***, const char *** *argv***);**

int irExecle (channel_id *cid***, const char *** *service***, const void *** *pdata***, int** *psize***, const char *** *path***, const char *** *arg0***,**

int irExecve (channel_id *cid***, const char *** *service***, const void *** *pdata***, int** *psize***, const char *** *path***, char *const *** *argv***, const char *** *envp***);**

int irExeclp (channel_id *cid***, const char *** *service***, const void *** *pdata***, int** *psize***, const char *** *file***, const char *** *arg0***,**

int irExecvp (channel_id *cid***, const char *** *service***, const void *** *pdata***, int** *psize***, const char *** *file***, char *const *** *argv***);**

Description

The *irExec* functions pass control of channel_id *cid* from the process invoking the *irExec* function to the process specified by *name* in the case of permanent processes, or *path* and *file* in the case of transient processes. The semantics of *irExec* differ slightly from the semantics of *exec*. In particular, the process invoking the *irExec* function is *not* replaced by the image of another process; the *irExec* call is return.

The *irExecp* function is used to pass control of the channel with *cid* to the permanent IRAPI process which used *name* as the argument to *irRegister* [see *irRegister(3IRAPI)*]. The process specified by the *name* receives a **IRE_EXEC** event [see *IrEVENTS(4IRAPI)*] for channel with *cid*. It must then *irInit(3IRAPI)* the channel with *cid*.

The *irExecl*, *irExecv*, *irExecle*, *irExecve*, *irExeclp*, and *irExecvp* functions are used to pass control of the *cid* to the transient IRAPI process specified by *path* or *file*. These functions fork and exec process *path* or *file* using the corresponding *exec(2)* system call with the same set of arguments passed to the *irExec* function. See *exec(2)* for more details on the arguments and differences between *exec(2)* calls. When the process calls *irRegister* upon startup, it receives a **IRE_EXEC** event for channel *cid*. It must then *irInit* the channel *cid*.

If the process that receives the **IRE_EXEC** event fails to call *irInit* for channel *cid* within a set time limit, the ownership of *cid* is returned to its default owner [see *irIntro(3IRAPI)*]. The default owner receives a IRE_DEFOWNER event for *cid*.

Before passing control of *cid*, the *irExec* functions set the **IRP_SERVICE_NAME** parameter for *cid* to be *service* [see *irParam(3IRAPI)* and *irPARAMETERS(4IRAPI)*]. IRAPI applications can use the **IRP_SERVICE_NAME** parameter to pass information. For example, the Application Dispatch (AD) process always passes the service name of the application to the *irExec* functions when passing control of *cid* in response to a new call.

A generalized data passing mechanism is provided by the *irExec* functions. A data buffer address *pdata* and its size *psize* can be passed as arguments to the *irExec* functions. The size of the data buffer must be **IRD_MAXDATABUFFER**, but the actual data contained within the buffer can be of any size less than or equal to **IRD_MAXDATABUFFER**. The data buffer can contain any data that a process would like to transfer to the new owner of *cid*. There is no limitation on the format of the data buffer as its contents is determined by the process invoking the *irExec* function. If no data is to be transferred, then *pdata* must be *NULL* and *psize* must be zero. The *irExec* functions use the *pdata* pointer to set the **IRP_EXEC_BUF** parameter and the *psize* argument to set the **IRP_EXEC_BUF_LEN** parameter. When a process receives an **IRE_EXEC** *event, it can use the irGetParam(3IRAPI)* function to check the **IRP_EXEC_BUF_LEN** parameter to see if there is any data in the **IRP_EXEC_BUF** parameter. If so, it can use the *irGetParamStr(3IRAPI)* function to access the data. The process must interpret the information contained in the data buffer.

Processes can also pass data across *irExec*'s using the *irSetParam(3IRAPI)* and *irGetParam(3IRAPI)* functions to set and get the IRP_REGISTER<*n*> parameters for channel *cid* [see *IrPARAMETERS(4IRAPI)*].

All events for *cid* on the queue of the process invoking the *irExec* function are passed to the new owner of *cid* with the sequence of the events preserved. Events for channel *cid* that occur between the time *irExec* is called and when the new process calls *irInit* are placed on the end of the event queue for channel *cid*.

The first event the process specified by *name, path,* or *file* receive is an **IRE_EXEC** event. The **IRE_EXEC** event fields contains **event_mod1** which is channel number for the *cid*

Return Value

The *irExec* functions return:

IRR_OK if the request is successful.

IRR_FAIL if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid argument is passed

IRER_NOTFOUND if the permanent process is not found for *irExecp*

IRER_BADSTATE if the cid is not in the IDLE state

IRER_INTERNAL if an internal error occurs

IRER_SYSERROR if a system error occurs (*irSysError* is set to the corresponding *errno*)

Caveat

Note that once the processing for the *irExec* functions progress so far, it is impossible for the *irExec* function to return failure. So there are cases where the *irExec* function returns success, but the requested action actually fails. An error message is placed in the error log if this occurs.

See Also

exec(2), irInit(3IRAPI), irIntro(3IRAPI), irParam(3IRAPI), irRegister(3IRAPI), IRAPI-AD(4IRAPI-AD), IrDEFINES(4IRAPI), IrEVENTS(4IRAPI), IrPARAMETERS(4IRAPI)

irFlash

Name

irFlash — Flash the switch-hook

Synopsis

#include <**irapi.h**>

int irFlash (channel_id *cid***, int** *tag***);**

Description

The *irFlash* function flashes the switch-hook for the channel specified as *cid*.

Tag is a user-supplied number that associates an *irFlash* function call with a subsequent event. The event is returned in an event structure *ir_event_struct* [see *IrEVENTS(4IRAPI)*] that is accessed by *irCheck(3IRAPI)*.

IRAPI library parameter IRP_FLASH_DURATION [see *IrPARAMETERS(4IRAPI)*] defines the duration of the switch-hook flash. Its existing value can be retrieved using *irGetParam(3IRAPI)* while a new value can be assigned to it using *irSetParam(3IRAPI)*. The default IRP_FLASH_DURATION depends on the parameters associated with switch integration. If a different flash duration is required, this parameter must be set to the desired value before invoking *irFlash*. *irFlash* uses the value available at the time of flashing the switch-hook.

IRAPI library parameter IRP_FLASH_TYPE indicates whether to wait after generating the flash. The default value IRD_FLASH_NO_WAIT applies to all telephony types for which flash is supported and indicates that the IRE_FLASH_DONE event is generated as soon as the flash has been generated. The optional value of IRD_FLASH_AND_WAIT applies only to Line Side T1 (LST1) and indicates that the library should wait briefly after generating the flash so that the switch can prepare to detecting DTMF digits used for dialing. Currently with LST1, there is no dial tone detection so this brief wait helps the switch to prepare to detect digits, but cannot ensure that the switch is ready.

The FLASH_DONE_EVENT indicates that the flash has been generated, but does not ensure that the switch is ready to accept digits for dialing. When used with Tip/Ring (T/R) channels, the application may wish to also wait for an IRE_CALL_PROG event indicating whether dial tone or some other tone is detected. (*event_mod1* is IREM_DIALTONE or IREM_STUTTER_DIALTONE when dialtone is detected). As mentioned above, dial tone detection is not available with LST1 so IRD_FLASH_AND_WAIT for IRP_FLASH_TYPE should be used instead.

WARNING: The parameters described above must be changed *before* calling *irFlash* or *after* receiving the IRE_FLASH_DONE event. Changing them while a flash attempt is in progress can lead to unexpected results.

Flashing is not supported for all telephony types (eg. PRI) and will only achieve the desired results if the switch supports flash in the manner desired.

Event

An IRE_FLASH_DONE event is returned with the appropriate modifier to indicate the success or failure of the function. Applications should be prepared to deal with the the dialtone event occurring either before or after the IRE_FLASH_DONE event.

Return Value

IRR_OK is returned if the request is successfully initiated.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_SERVICESTATE if the *cid* is not in the IRD_ACTIVE service state

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_UNSUPPORTED if a flash is attempted on telephony hardware that does not support the feature

IRER_BADSTATE if the *cid* is not in the IRS_IDLE library state

See Also

irSetParam(3IRAPI), irGetParam(3IRAPI), IrPARAMETERS(4IRAPI), IrEVENTS(4IRAPI)

irFlushInput

Name

irFlushInput — Remove all characters from the input queue

Synopsis

#include <**irapi.h**>

int irFlushInput (channel_id *cid***);**

Description

The *irFlushInput* function removes all characters from the input queue for the specified channel. Since input characters are always being collected, *irGetInput(3IRAPI)* might retrieve unwanted characters left in the input queue. *irFlushInput* can be used to prevent dial-ahead. The application starts collecting input characters at a known point in time (that is, when *irFlushInput* is called).

Event

No event results from the call to *irFlushInput*.

Return Value

The number of characters flushed is returned if *irFlushInput* is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

See Also

irGetInput(3IRAPI), irUngetInput

irForceInit

Name

irForceInit — Forcibly obtain a channel, initialize a channel, and obtain a channel ID

Synopsis

#include <**irapi.h**>

int irForceInit (int *channel_nbr***, channel_id ***_cid_ptr_ **, int** *tag***);**

Description

The *irForceInit* function unconditionally obtains ownership and initializes a channel specified by a *channel_nbr*. This function is similar to *irInit(3IRAPI)*, but does not permit the previous owner to retain ownership of a channel. This function is reserved for use by maintenance and related processes (such as MTC_PROXY) and should not be used by applications.

Except upon failure, this function sets **cid_ptr* to the *channel_id* that is to be used by most of the other functions when referencing the channel.

irForceInit sets all channel parameters to their appropriate default values. "Save on Exec" parameters are set to their default values, unlike *irInit(3IRAPI)* which preserves parameter values on initialization.

If *irForceInit* returns IRR_OK, the channel is in the IRS_IDLE state [see *IrSTATES(4IRAPI)*]. If *irForceInit* returns IRR_PENDING, the channel is in the IRS_INIT_PENDING state.

A maintenance process can unconditionally request a channel from an application with *irForceInit*. The application that previously owned the channel should discontinue use of the *channel_id* as the channel is invalid.

If the channel is not allocated at the time of the request, the requesting maintenance process receives a return value of IRR_OK and **cid_ptr* is set to the *channel_id* that should be used with other functions.

If the channel is allocated at the time of the request, the requesting maintenance process receives a return value of IRR_PENDING and **cid_ptr* is set to the *channel_id* that should be used with other functions. In this case, the owning process, acting on behalf of the library, frees the channel and passes ownership to the requesting process. The requesting process experiences a slight delay before receiving the IRE_GRANT event. The current owner of the channel cannot control whether the channel ownership passes from itself to the requester with the IRP_CHAN_NEGOTIATION parameter. The current owner is informed of the channel deallocation through the IRE_CHAN_REMOVED event.

Event

The IRE_GRANT event is returned from a call to *irForceInit* in which the return value is IRR_PENDING.

Return Value

IRR_OK is returned on success.

IRR_PENDING is returned if the channel has not yet been released by the prior application. The **cid_ptr* is valid but no other functions may be called until the IRE_GRANT event is received to indicate a state change from IRS_INIT_PENDING to IRS_IDLE.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *channel_nbr* specifies an invalid channel or if the *cid_ptr* is NULL

IRER_NOREGISTER if the processes has not previously called

IRER_PREVIOUS_INIT if the channel has already been initialized. If a process attempts to initialize a channel it already has a cid for, but does not think it should, it should *irDeinit(3IRAPI)* the cid.

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

Example

Note the use of "&" before the *cid* argument to *irForceInit* in the following example:

```
int channel_nbr;
channel_id cid;
int retval;
int tag;

   ...

retval = irForceInit(channel_nbr, &cid, tag);
switch (retval) {
    case IRR_FAIL:
        <report failure as appropriate>
        break;
    case IRR_OK:
        irChanOOS(cid);
        break;
    case IRR_PENDING:
        <wait for IRE_GRANT event>
        break;
}

   ...
```

See Also

irDeinit(3IRAPI), irInit(3IRAPI), irChanOOS(3IRAPI).

irFreeResource

Name

irFreeResource — Free previously reserved resource(s)

Synopsis

#include <**irapi.h**>

int irFreeResource (channel_id *cid***, ir_reserve_t *****presources***, int *****failure***);**

Description

The *irFreeResource* function deallocates one or more resources, previously reserved with *irReserveResource*, on the channel identified by *cid*. *presources* is an array of capabilities/implementation pairs terminated by the IRC_NULL capability. Each capability/implementation pair in the array indicates a resource to be freed. If implementation is set to IRD_INVALID for any capability, all implementations of the indicated capability are freed. Once freed, the resource is available for use by other channels.

If *presources* is a NULL pointer, then all resources currently allocated by the *cid* are freed.

Event

No event results from the call to *irFreeResource*.

Return Value

IRR_OK is returned if *irFreeResource* is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_RESOURCEBUSY if some resource in the resource list is currently in use *failure* is the index of the offending resource in *presources*. All resources prior to the offending element are freed. All resources past the offending element, including the offending element, remain allocated. When *presources* is NULL, and some resource previously allocated is in use, *failure* is not set. Also, no previously allocated resources are freed.

irFreeResource ignores the request to free non-allocated resources, and ignores any invalid capability/implementation pairs in the list.

See Also

irReserveRes(3IRAPI) and *IrRESOURCES(4IRAPI)*.

irGetAlgorithm

Name

irGetAlgorithm, irFGetAlgorithm, irBGetAlgorithm — Get the coding algorithm used in voice object

Synopsis

#include <**irapi.h**>

int irGetAlgorithm (vf_descriptor *vfd***);**

int irFGetAlgorithm (const char ***_voice_file_);**

int irBGetAlgorithm (const char ***_voice_buf_, int** *size);*

Description

The *irGetAlgorithm* function returns the speech coding algorithm contained in the voice file specified by the *vfd*.

irFGetAlgorithm returns speech coding algorithm contained in the voice file *voice_file*.

irBGetAlgorithm returns speech coding algorithm contained in the voice buffer *voice_buf* for the length, in bytes, specified by the *size* argument.

Return Value

All these functions return the name of the algorithm [see *IrALGORITHMS(4IRAPI)*] if successful.

IRR_FAIL is returned if an error occurs.

Errors

irError is set as follows if an error occurs:

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_INVALID if the *vfd* is invalid, if the *voice_buf* is the NULL pointer or if *size* is not greater than *sizeof(short) * 2*

IRER_UNSUPPORTED if the data object was coded with an algorithm which could not be recognized. This error is also returned if the object was some arbitrary data such as text.

See Also

irLSeek(3IRAPI), irTime2Byte(3IRAPI), irByte2Time(3IRAPI), IrALGORITHMS(4IRAPI)

irGetInput

Name

irGetInput — Collect an input string

Synopsis

#include <**irapi.h**>

int irGetInput (channel_id *cid***, char ***i_buf***, int** *length***);**

Description

The *irGetInput* function copies the *length* input characters found in the input queue to *i_buf* for the specified telephone channel (*cid*). The string is terminated with '\0' (null character) if the length of the input is less than *length*. If the length of the input is greater than or equal to *length*, the string will not be terminated with a null character. If the length of the input queue is greater than *length*, the remaining characters are left on the queue.

The input queue is the facility through which all caller supplied input is provided to the application. Input includes touch-tone input and speech recognition input. Multi-character input, such as input provided via FlexWord speech recognition, appears as a sequence of input on the input queue. In this sense, the input queue can also be thought of as an input stream.

irGetInput is typically used after a IRE_INPUT_DONE event is received indicating that a string has been collected. The IRE_INPUT_DONE event is generated under the following conditions controlled by IRAPI library parameters [see *IrPARAMETERS(4IRAPI)*] for input collection:

- Parameter IRP_INPUT_LEN specifies the number of sequential inputs that must be received before IRE_INPUT_DONE event is generated with an event modifier of IREM_INPUT_LENGTH.
- Parameter IRP_INPUT_DELIM1 or IRP_INPUT_DELIM2 specifies a delimiter that, when received as input immediately, causes IRE_INPUT_DONE event to be generated with an event modifier of IREM_INPUT_DELIM.

Calling *irGetInput* with *length* as 0 causes *irGetInput* to return the number of characters currently on the input queue. No characters are removed from the input queue.

Event

No event results from the call to *irGetInput*.

Return Value

The number of characters written to *i_buf* if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

See Also

irUngetInput(3IRAPI), irFlushInput(3IRAPI), irStartTTTimer(3IRAPI), irStart(3IRAPI) and IrEVENTS(4IRAPI)

irGetQKey

Name

irGetQKey — Get the queue key of a specified process

Synopsis

#include <**irapi.h**>

ir_key_t irGetQKey (char *****name***);**

Description

The *irGetQKey* function returns the message key associated with the process *name*. It is assumed that *name* is a IRAPI application which has been registered with the voice system via *irRegister* and *name* matches the name with which it was registered.

The Qkey is the medium though which all non-channel based message passing occurs.

Event

No event is generated as a result of a call to *irGetQKey*.

Return Value

The QKey of the calling process is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_NOTFOUND if the *name* does not have a QKey

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

See Also

irPostEvent(3IRAPI)

irGetVCount

Name

irGetVCount — Return the voice stream count after an event

Synopsis

#include <**irapi.h**>

long irGetVCount (channel_id *cid***);**

Description

The *irGetVCount* function returns the amount of time played or recorded when an IRE_PLAY_DONE, an IRE_PLAY_PROG or an IRE_RECORD_DONE event occurs for the channel indicated by *cid*.

Time is accumulated from the most recent *irEnd(3IRAPI)*, *irRecord(3IRAPI)*, *irPlayResume(3IRAPI)* or *irRecordResume(3IRAPI)*.

Event

No event results from the call to *irGetVCount*.

Return Value

The time value in milliseconds is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if the *cid* is invalid.

Example

The count is the amount of time *played* from the starting point to the point of interruption or to the end of the voice file. This is not the same as position, which is the starting point plus the time played or recorded.

A typical calling sequence is

```
irPlay( cid, 1, file, 0 );
irEnd( cid, 0 );
irWait( );  /* block until event */
event_id = irCheck( event );  /* get event */
if( event_id == IRE_PLAY_DONE && event->cid == cid ) {
      time_played = irGetVCount( event->cid )
}
```

See Also

irLSeek(3IRAPI), irByte2Time(3IRAPI), irTime2Byte(3IRAPI)

irGlobalParam

Name

irGetGlobalParam, irGetGlobalParamStr — Get the value of one or more IRAPI library global parameters

Synopsis

#include <**irapi.h**>

int irGetGlobalParam (int *identifier***, int ****value***);**

int irGetGlobalParamStr (int *identifier***, char ****value***, int** *count***);**

Description

The *irGetGlobalParam* and *irGetGlobalParamStr* functions get the current value of an IRAPI global parameter. See *irAPI.rc(4IRAPI)* for a list of valid parameters and their default and legal values.

irGetGlobalParamStr copies, at most, *count* bytes into *value*. The application developer must verify that *value* is large enough; the number of bytes required can be found by calling *irGetGlobalParamStr* with *count* set to 0.

Global parameters are system-wide and are the same for every application.

Parameter		type	size
Valid Values	Describes values to which this parameter may be set		
Default Value	Describes the default value to which this parameter will be set		
Description	Describes how and where the parameter is used		
	IRP_DNISWAIT	int	4
Valid Values	Any integer between 0 and 60		
Default Value	2		
Description	This value is the maximum number of seconds the IRAPI library waits for the DNIS to arrive from the switch.		

	IRP_ANALOG_LOSS_COMP	int	4
Valid Values	Integers between -18 and +3 in 3 dB increments		
Default Value	0		
Description	This value is the amount of gain (in dB) to be added to standard gain adjustment for a bridge that includes an analog (T/R) channel.		

	IRP_DIGITAL_LOSS_COMP	int	4
Valid Values	Integers between -15 and +6 in 3 dB increments		
Default Value	0		
Description	This value is the amount of gain (in dB) to be added to standard gain adjustment for a bridge that includes only digital channels.		

	IRP_BACKGROUND_OVOL	int	4
Valid Values	Any integer between 0 and 100		
Default Value	33		
Description	This value is the adustment to OVOL (output volume level) in percent for speech being played in background.		

	IRP_VCHANS	int	4
Valid Values	Any integer between 1 and 96		
Default Value	1		
Description	This value is the the number of virtual channels to be allocated on the system.		

	IRP_AD_READ_ONLY	int	4
Valid Values	Any integer		
Default Value	0		
Description	A value of zero indicates that processes are allowed to use the IRAPI Application Dispatch (AD) library routines to change the AD Tables. The value of non-zero indicates that processes NOT are allowed to use the IRAPI-AD library routines to change the AD Tables.		

	IRP_TRACE_BUFFER_SIZE	int	4
Valid Values	Any integer between 128 and 40000		
Default Value	4000		
Description	This value is the number of trace messages in the trace buffer. If trace messages are being lost, this value can be increased at the expense of system memory.		
	IRP_NON_AD_CHAN_LIST	char	256
Valid Values	Comma or colon separated list of channel numbers or channel ranges		
Default Value	Null string		
Description	This value is a list of channels for which the AD process should not be the default channel owner.		
	IRP_AD_CHANNEL_TABLE	char	128
Valid Values	Any valid pathname		
Default Value	/vs/data/ad_channel_table		
Description	This value is the full path file name for the channel table used by the AD library.		
	IRP_AD_DNISANI_TABLE	char	128
Valid Values	Any valid pathname		
Default Value	/vs/data/ad_dnisani_table		
Description	This value is the full path file name for the DNIS/ANI table used by the AD library.		
	IRP_SPEECHDIR	char	128
Valid Values	Any valid pathname		
Default Value	/home2/vfs/talkfiles		
Description	This value is the full path directory name for the speech filesystem used by IRAPI library.		

Return Value

irGetGlobalParamStr returns the number of bytes copied to *value* if *count* is positive. If *count* is 0, the number of bytes that would have been copied to *value* is returned.

irGetGlobalParam returns IRR_OK if successful.

All functions return IRR_FAIL if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *identifier* is not valid, if the *count* is negative, or if the *value* is not valid for *identifier*

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

See Also

irAPI.rc(4IRAPI).

irHBridge

Name

irHBridge — Add/drop a half-bridge to another channel

Synopsis

#include <**irapi.h**>

int irHBridge (channel_id *cid1***, int** *chan2***, int** *flag***);**

Description

The *irHBridge* function half-bridges channel *cid1* to *chan2* or drops an existing half-bridge between the two channels. Half-bridging means that the audio coming in on *chan2* is heard on *cid1*.

This function is achieved by low level timeslot arbitration functions.

Value of *flag* determines if the half-bridge should be added or dropped. Use IRD_ADD [see *IrDEFINES(4IRAPI)*] as *flag* to create the half-bridge. To drop the half-bridge, *irHBridge* must be invoked with IRD_DROP as the value of *flag*.

The half-bridge is dropped when *cid1* is released via *irDeinit(3IRAPI)* by its owning process.

The IRAPI supports the CONVERSANT VIS Transmission Level Plan (TLP) for call bridging. Refer to *Intuity CONVERSANT VIS V5.0 Communication Development*, 585-310-229, for information about the CONVERSANT VIS TLP.

The IRP_DTMF_MUTING parameter determines whether the input of touch-tone digits is passed through the bridge. When bridging to another automated response system, IRP_DTMF_MUTING must be set to IRD_ON to allow touch-tone digits to pass through the bridge. DTMF muting is automatically enabled on Tip/Ring (T/R) cards and can currently only be controlled on digital interfaces (T1, PRI, and LST1)

Each channel can have at most 7 output timeslots of which one is pre-assigned. Therefore, channel can only add another 6 output timeslots.

Event

No event results from a call to *irHBridge*.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if invalid arguments are passed

IRER_MAXCHAN_TIMESLOTS if the channel has reached the limit for output timeslots (7 maximum per channel)

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_TSBUS if an attempt is made to half-bridge between two channels on buses which do not communicate or if either channel is not on a bus

See Also

irMonitor(3IRAPI)

Caveat

Note that the half-bridge to channel *chan2* stays active until dropped regardless of the state of channel *chan2*.

irIE

Name

irSetIE, irSetIEs, irGetIE, irGetIEs — Set/get an information element

Synopsis

#include <**irapi.h**>

int irSetIE (channel_id *cid***, int** *info_ele***, int** *value***);**

int irSetIEs (channel_id *cid***, int** *info_ele***, char *** *value***);**

int irGetIE (channel_id *cid***, int** *info_ele***, int *** *value***);**

int irGetIEs (channel_id *cid***, int** *info_ele***, char *** *value***);**

Description

On some trunk connections, information describing incoming or outgoing calls is available as "information elements." The information elements available depends on the telephony type and the provisioning of the switch that connects to the VIS.

Information elements are internally maintained by the IRAPI library as a part of a call profile. The *irSetIE* and *irSetIEs* functions set the value of an information element in the call profile while *irGetIE* and *irGetIEs* get the value of an information element from the call profile.

The *cid* is the channel identifier. The *info_ele* is the name of the information element whose value is to be set or retrieved. In *irSetIE*, *value* contains the new integer value to which *info_ele* should be set. In *irGetIE*, *value* is used by IRAPI library to return the current integer value of *info_ele*.

Currently supported elements for incoming calls are IRD_ANI (automatic number identification), IRD_DNIS (dialed number identification service), IRD_REDIRECTING (redirecting number), and IRD_INBOUND_SERVICE (inbound service type).

IRD_ANI provides the Calling Party Number (CPN) which can be either the Station ID (SID) or the ANI number depending on the configuration of the switch (and/or the attributes associated with a service that requests the CPN on a call-by-call basis). IRD_DNIS provides the specific telephone number dialed by the caller or the routing number generated by the network when forwarding calls dialed to an 800 or 900 service number. IRD_REDIRECTING provides the number that a caller originally dialed before it was intercepted by the network and redirected to the VIS. IRD_INBOUND_SERVICE provides the service type for the inbound call. The possible values are defined as SVC_* in *tas_defs.h*.

If *irSetIEs* is called with *info_ele* set as IRD_ANI [see *IrDEFINES(4IRAPI)*], the ANI information element in the call profile will be set to the value specified in *value* while if it is called with *info_ele* set as IRD_DNIS, the DNIS information element in the call profile will be set to the value specified in *value*.

If *irGetIEs* sets *info_ele* to IRD_ANI, the IRD_MAX_ANI_DIGITS address digits received during the setup of the last incoming call for the *cid* are copied into *value*. If *irGetIEs* sets *info_ele* to IRD_DNIS, the IRD_MAX_DNIS_DIGITS address digits received during the setup of the last incoming call for the *cid* are copied into *value*. If *irGetIEs* sets *info_ele* to IRD_REDIRECTING, the IRD_MAX_REDIRECTING_DIGITS are copied into *value*. The application developer must declare *value* to be at least IRD_MAX_ANI_DIGITS, IRD_MAX_DNIS_DIGITS, or IRD_MAX_REDIRECTING_DIGITS in length depending upon the information element required.

The *irSetIE* and *irGetIE* functions are used to set or get the integer value for IRD_INBOUND_SERVICE.

irSetIE and *irSetIEs* are typically used internally by the IRAPI library (or startup services using the converse vector step) to set the information elements for incoming calls. *irGetIE* and *irGetIEs* are typically used by the Application Dispatch (AD) process or by applications after an IRE_NEWCALL event is received to indicate that a new call has arrived.

The information elements associated with incoming calls are overwritten by the next incoming call and this can occur before an application has called *irDisconnect* or *irDeinit*. Therefore, an application that wishes to record the incoming information elements, such as in call data records, must obtain the information elements before calling *irDisconnect* or *irDeinit* and before calling *irWait* after an IRE_DISCONNECT event. The information elements for incoming calls are undefined as soon as the incoming call is terminated and while an outbound call is taking place.

In addition to the information elements associated with incoming calls, there are information elements associated with outgoing calls.

irSetIE and *irGetIE* may be used to modify or get the IRD_SERVICE_TYPE information element which is of integer type. This information element can be used to select the service type on outbound calls. The available service types are defined as SVC_* in *tas_defs.h*. The default value IRD_INVALID can be used if there is no need to select a service type.

irSetIE and *irGetIE* may be used to modify or get the IRD_BEARER_CAP information element which is of integer type. This information element can be used to specify the bearer capability on outbound calls. The available bearer capability types are defined as BEAR* in *tas_defs.h*. The default value IRD_INVALID can be used if there is no need to select a bearer capability.

irSetIEs and *irGetIEs* may be used to modify or get the IRD_OUTBOUND_ANI information element which is of buffer type. This information element can be used to set the ANI on outbound calls. If no value is provided, a NULL string is used by default.

Information elements associated with outbound calls must be set before the call is initiated with *irCall*. These information elements are returned to their default values when the channel is released via *irDeinit*.

Information elements are preserved across *irExec(3IRAPI)* boundaries.

Event

No event results from calling these functions.

Return Value

irSetIE returns IRR_OK if successful.

irGetIE returns the number of characters written to *value* if successful.

IRR_FAIL is returned if an error occurs with any of the functions.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* or *info_ele* are invalid

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

Caveats

irGetIEs puts a null string in *value* if the corresponding information (ANI, DNIS, etc.) is not available for the last call received on *cid*.

See Also

IrDEFINES(4IRAPI)

irInit

Name

irInit, irInitCancel — Initialize a channel and obtain a channel ID

Synopsis

#include <**irapi.h**>

int irInit (int *channel_nbr***, channel_id ***cid_ptr***, int** *return_mode***, int** *tag***);**

int irInitCancel (channel_id *cid***);**

Description

The *irInit* function initializes a channel specified by *channel_nbr*.

Except when it fails, this function sets **cid_ptr* to the *channel_id* that is to be used by most of the other functions when referencing the channel.

return_mode indicates how the caller expects channel allocation to be handled. Allocation of channels behaves much like the allocation of other resources. The behavior determined by *return_mode* is summarized as follows:

IRD_IMMEDIATE	Indicates *irInit* should return immediately with IRR_FAIL if the channel cannot be immediately allocated.
IRD_BLOCKFOREVER	Indicates *irInit* should return IRR_PENDING if the channel cannot be immediately allocated. When IRR_PENDING is returned, the calling application should be prepared to deal with a subsequent IRE_CHAN_GRANT event.
Any positive integer *N*	Indicates *irInit* should return IRR_PENDING if the channel cannot be immediately allocated. When IRR_PENDING is returned, the calling application should be prepared to deal with a subsequent IRE_CHAN_GRANT event or a subsequent IRE_CHAN_DENY event after *N* milliseconds.

tag is returned to the application with the subsequent IRE_CHAN_GRANT or IRE_CHAN_DENY event.

irInit's behavior is affected strongly by the current ownership of *channel_nbr* and the intent of the function call in the following two ways:

First, an application can request a channel from another process with *irInit*. If the channel is currently owned by another process, applications should expect a return code of IRR_PENDING, and therefore *return_mode* should be set to something other than IRD_IMMEDIATE. There is no hope of acquiring a channel from another process if the application is not willing to wait. The current owner indicates the availability of the channel by setting the IRP_CHAN_NEGOTIATION parameter. See subsequent paragraphs and *IrPARAMETERS(4IRAPI)* for more information on IRP_CHAN_NEGOTIATION. When a channel is owned by its default owner, in particular the Application Dispatch (AD) process, it is likely that the channel is granted as quickly as the library can inform AD that the channel has been removed. *irInit* still returns IRR_PENDING even in this case. Acquiring channel ownership in this manner is used typically, but not necessarily, to initiate an outbound call.

Second, applications should call *irInit* in response to IRE_EXEC events to acquire ownership for the channel. The call to *irInit* informs the library that the application is ready to take ownership of the channel. Since the library is ready to hand the channel to the calling processes, IRR_PENDING is not returned and *return_mode* may be set to IRD_IMMEDIATE without denial.

If *irInit* returns without error, *irInit* sets all channel parameters to their appropriate "save on exec" or default values. Some channel parameters are "saved on exec" as indicated in *IrPARAMETERS(3IRAPI)* and are preserved when the channel ownership has been transferred via a call to one of the *irExec(3IRAPI)* functions. The remaining parameters are set to the appropriate default value. When transfer of ownership has not occurred as the result of an *irExec(3IRAPI)*, all parameters are set to their appropriate defaults.

If *irInit* returns IRR_OK, the channel is in the IRS_IDLE state [see *IrSTATES(4IRAPI)*]. If *irInit* returns IRR_PENDING, the channel is in the IRS_INIT_PENDING state. No voice operations are accepted for a *channel_id* in the IRS_INIT_PENDING state.

The application that owns the channel can control how the channel ownership passes from itself to the requester with the IRP_CHAN_NEGOTIATION parameter. This parameter can take two values: IRD_ALLOW (to allow all requests unconditionally) and IRD_CONDITIONAL (to allow the application to process the request and make a decision for itself).

If the owning application is always willing to give up the channel (that is, IRP_CHAN_NEGOTIATION is set to IRD_ALLOW), the IRAPI grants the channel to the requester, and sends a IRE_CHAN_REMOVED event to the former owner indicating that the channel has been deallocated. The application that previously owned the channel does not need to take any action to transfer ownership of the channel; however, that application must discontinue use of the *channel_id* it had been using for the channel.

An application that sets IRP_CHAN_NEGOTIATION to IRD_CONDITIONAL receives an IRE_CHAN_REQUESTED event when the channel it owns is requested. The application may choose to free the channel via *irDeinit(3IRAPI)* or ignore the event and continue to use the channel. Part of the *irDeinit(3IRAPI)* processing includes generating an IRE_CHAN_GRANT event for the requester indicating that the channel has been granted to it.

Regardless of the setting for IRP_CHAN_NEGOTIATION, if the channel is active as indicated by the library and service states, it is not taken from the current owner.

The *irInitCancel* function cancels a channel or group initialization request on a *cid* that is in the IRS_INIT_PENDING state. If the channel has already been granted, *irInitCancel* returns IRR_FAIL with *irError* set to IRER_BADSTATE. This condition may occur if the channel was granted to *irWait(3IRAPI)* between calls. The application should call *irDeinit(3IRAPI)* upon receipt of IRE_CHAN_GRANT if ownership of the channel is no longer required.

The *cid* becomes invalid when *irInitCancel* is successful.

Multiple processes may compete for ownership of a channel. Channel ownership is granted on a first-come-first-served basis. Requests via *irInit* have priority over requests made via *irInitGroup(3IRAPI)*.

Event

The IRE_CHAN_GRANT event occurs if a call to *irInit* returns an IRR_PENDING value.

The IRE_CHAN_DENY event occurs if the channel cannot be granted within the time specified in the *return_mode* argument. The application should discontinue use of the *channel_id* after receiving the IRE_CHAN_DENY event.

Return Value

IRR_OK is returned if the request is successful.

IRR_PENDING is returned if the channel has not yet been released by the prior application. The **cid_ptr* is valid but no other functions may be called until the IRE_CHAN_GRANT event is received. This event indicates a state change from IRS_INIT_PENDING to IRS_IDLE.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *channel_nbr* specifies an invalid channel, *return_mode* is invalid, or *cid_ptr* is NULL

IRER_NOREGISTER, for *irInit* only, if the process has not previously called *irRegister(3IRAPI)*

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_RESOURCEBUSY if the channel is currently busy and *return_mode* is set to IRD_IMMEDIATE

IRER_BADSTATE, for *irInitCancel*, if the channel is not in the IRS_INIT_PENDING state or if the channel has been granted between calls to *irWait(3IRAPI)*

Example

The following example is how to initialize a channel. Note the use of "&" before the *cid* argument to *irInit* in the following example:

```
int channel_nbr;
channel_id cid;
int retval;
int tag;

   ...

retval = irInit(channel_nbr, &cid, IRD_BLOCKFOREVER, tag);
switch (retval) {
    case IRR_FAIL:
         <report failure as appropriate>
         break;
    case IRR_OK:
         irCall(cid, ...)
         break;
    case IRR_PENDING:
         <wait for IRE_CHAN_GRANT or IRE_CHAN_DENY event>
         break;
}

   ...
```

See Also

irDeinit(3IRAPI), *IrPARAMETERS(4IRAPI)*, *IrEVENTS(4IRAPI)*.

Caveat

Note: It is possible to configure a system where incoming and outgoing calls use the same set of ports. In this situation, an application can get a pseudo-glare condition, that is, an outbound application has acquired a channel, but has not yet initiated an outbound call. In this interval, a new call can arrive. The ‘‘right’’ behavior for an application is to release the channel and allow the incoming call to proceed. The application should *irExec* the channel back to the default owner, which is AD by default. The default owner is responsible for dispatching a new application.

irInitGroup

Name

irInitGroup — Initialize a channel and obtain channel id from an equipment group

Synopsis

#include <**irapi.h**>

int irInitGroup (int *equip_grp***, channel_id ***_cid_ptr_**, int** *return_mode***, int** *tag***);**

Description

The *irInitGroup* function initializes a channel in equipment group *equip_grp*.

Except upon failure, this function sets **cid_ptr* to the *channel_id* that is to be used by most of the other functions when referencing the channel.

return_mode indicates how the caller of the function expects channel allocation to be handled. Allocation of channels behaves much like the allocation of other resources. The behavior determined by *return_mode* is summarized as follows:

IRD_BLOCKFOREVER	Indicates *irInitGroup* should return IRR_PENDING if the channel cannot be immediately allocated. When IRR_PENDING is returned, the calling application should be prepared to deal with a subsequent IRE_CHAN_GRANT event.
Any positive integer *N*	Indicates *irInitGroup* should return IRR_PENDING if the channel cannot be immediately allocated. When IRR_PENDING is returned, the calling application should be prepared to deal with a subsequent IRE_CHAN_GRANT event or a subsequent IRE_CHAN_DENY event after *N* milliseconds.

tag is returned to the application with the subsequent IRE_CHAN_GRANT or IRE_CHAN_DENY event.

irInitGroup selects as negotiable channels in *equip_grp* only if they are currently owned by the default owner (usually the Application Dispatch process). The library determines the state of the channel and releases it to the requesting application if the channel is not busy.

If no negotiable channels are found, the first channel in *equip_grp* freed by some process is allocated to the requester.

Channels are allocated to requesters on a first-come-first-served basis. Channel requests made via *irInit(3IRAPI)* have priority over those made via *irInitGroup*.

The equipment group *equip_group* can be any of the administerable equipment groups 0 to 31, or one of the two following abstract equipment groups. IRD_REAL_CHANS represents any of the real channels in the system and IRD_VIRTUAL_CHANS represents any of the virtual channels in the system.

Event

The IRE_CHAN_GRANT event occurs if the return value is IRR_PENDING.

The IRE_CHAN_DENY event occurs if a channel from *equip_grp* cannot be granted within the time specified in the *return_mode* argument. The application should discontinue use of the *channel_id* after receiving the IRE_CHAN_DENY event.

Return Value

IRR_PENDING is returned if a channel from *equip_grp* has not yet been released by the prior application. The **cid_ptr* is valid, but no other functions may be called until the IRE_CHAN_GRANT event is received. This event indicates a state change from IRS_INIT_PENDING to IRS_IDLE.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if *equip_grp* specifies an invalid equipment group, *return_mode* is invalid, or *cid_ptr* is NULL

IRER_NOREGISTER if the process has not previously called *irRegister(3IRAPI)*

IRER_RESOURCEBUSY if *return_mode* is IRD_IMMEDIATE and there is no channel immediately available

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information.)

Example

The following examples shows how to initialize a group. Note the use of "&" before the *cid* argument to *irInitGroup* in the following example:

```
int equip_grp;
channel_id cid;
int retval;
int tag;

   ...

retval = irInitGroup(equip_grp, &cid, IRD_BLOCKFOREVER, tag);
switch (retval) {
    case IRR_FAIL:
         <report failure as appropriate>
         break;
    case IRR_PENDING:
         <wait for IRE_CHAN_GRANT or IRE_CHAN_DENY event>
         break;
}

   ...
```

See Also

irDeinit(3IRAPI), *irInit(3IRAPI)*, *IrPARAMETERS(4IRAPI)*, *IrEVENTS(4IRAPI)*.

Caveat

Note that it is possible to configure a system where incoming and outgoing calls occur over the same set of ports. In this situation, an application can get a pseudo-glare condition: an outbound application has acquired a channel, but has not yet initiated an outbound call. In this interval, a new call can enter the system. The ‘‘right’’ behavior for an application is release the channel and allow the incoming call to proceed. The application should *irExec(3IRAPI)* the default owner on the channel, which is AD by default. The default owner is responsible for dispatching a new application.

irLBolt

Name

irLBolt, irNap — UNIX clock routines

Synopsis

#include <**irapi.h**>

int irLBolt ( );

int irNap (int *interval***);**

Description

The *irLBolt* function returns the *lbolt* time for the system. The lbolt time is defined as the amount of time which has passed since the UNIX system was last booted. The unit of time is hundredths of seconds.

The *irNap* function sleeps for *interval* hundredths of seconds. This is a blocking sleep, therefore, unlike most other IRAPI functions, it does not return immediately. Also, there are no events indicating the completion of *irNap*.

Events

No event results from the call to either of these functions.

Return Value

irLBolt, on success, returns the elapsed time since the last UNIX system reboot measured in hundredths of seconds. IRR_FAIL is returned if an error occurs.

irNap returns IRR_FAIL if an error occurs, otherwise the number of ‘‘unslept’’ hundredths of seconds are returned; 0 indicating ‘‘sleep’’ to completion, a positive value indicating interruption due to a signal.

Error

irError is set to IRER_SYSERROR if system call failure occurs (see *irSysError* for additional information).

Caveat

Since *irNap* is a blocking function, it should not be used in typical IRAPI applications (see *irIntro(3IRAPI)* for a discussion about potential problems with IRAPI applications which block for long periods of time).

These functions report values based on the inherent clock granularity of the currently supported hardware platform. This granularity is not guaranteed with future hardware platforms.

See Also

irTimer(3IRAPI), *irTTTimer(3IRAPI)*, *irRecogTimer(3IRAPI)*.

irLP

Name

irSpeakNum, irSpeakChar — Functions for lingual playback (LP) support

Synopsis

#include <**irapi.h**>

int irSpeakNum (channel_id *cid***, int** *integer***, int** *inflection***, int** *tag***);**

int irSpeakChar (channel_id *cid***, const char *****string***, int** *inflection***, int** *tag***);**

Description

These functions provide lingual playback (LP) support by speaking numeric or character phrases in the correct manner.

The *irSpeakNum* function speaks an integer *number* with the correct *inflection*s while *irSpeakChar* speaks a alpha-numeric string *string* with the correct *inflection*. Supported *inflection*s are IRD_INFLECTION_NONE, IRD_INFLECTION_RISING, IRD_INFLECTION_FALLING and IRD_INFLECTION_TOTAL [see *IrDEFINES(4IRAPI)*]. IRD_INFLECTION_TOTAL produces rising inflection on the first phrase and falling inflection on the last phrase if there are more than one. IRD_INFLECTION_NONE produces no inflection.

These functions assume *standard* speech files from UNIX file system directories are defined by the IRP_SPEECHDIR and IRP_TALKFILE parameters.

The channel moves to the IRS_PLAY_QUEUED state if these functions complete successfully. Multiple *irLP* functions may be queued, intermixed with *irPlay(3IRAPI)* functions. Play actually starts when *irEnd(3IRAPI)* is called.

Event

Since these functions are abstractions of *irPlay(3IRAPI)* and its variants, the play does not actually start until *irEnd(3IRAPI)* is called. An IRE_PLAY_DONE and possibly multiple IRE_PLAY_PROG events are then generated with the appropriate modifiers [see *IrEVENTS(4IRAPI)*].

Return Value

The number of phrases required to accomplish the request is returned if these functions are successful. This may be useful for applications counting IRE_PLAY_PROG events.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* or *inflection* is not valid

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_BADSTATE if the channel is not in the IRS_IDLE or IRS_PLAY_QUEUED state

These functions do not check for the existence of standard speech. If a voice file is not found, an error is logged by the voice response output process (VROP).

See Also

IrDEFINES(4IRAPI), IrPARAMETERS(4IRAPI)

irLSeek

Name

irLSeek — Move read/write pointer in voice file

Synopsis

#include <**irapi.h**> **#include** <**unistd.h**>

long irLSeek (vf_descriptor *vfd***, long** *offset***, int** *whence***);**

Description

The *irLSeek* function allows millisecond level access to a voice file referenced by an open voice file descriptor (*vfd*). A *vfd* is obtained by first calling *irOpen(3IRAPI)*. This function is similar to *lseek(2)*.

If *whence* is SEEK_SET, the pointer is set to *offset* milliseconds.

If *whence* is SEEK_CUR, the pointer is set to its current location plus the *offset*.

If *whence* is SEEK_END, the pointer is set to the length of the voice file plus the *offset*.

This function may be used to return the current position by setting *whence* to SEEK_CUR, and *offset* to 0.

Note: If a call to *irLSeek* results in a negative file pointer, *irLSeek* sets the file pointer to 0.

Event

No event results from the call to *irLSeek*.

Return Value

This function returns a non-negative long indicating the voice file pointer location in milliseconds.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if *vfd* is not an open voice file descriptor or *whence* is not SEEK_SET, SEEK_CUR, or SEEK_END

See Also

lseek(2), irGetVCount(3IRAPI)

irLibState

Name

irLibState — Return the library state of a channel

Synopsis

#include <**irapi.h**>

int irLibState (channel_id *cid***);**

Description

This function returns the library state associated with the *cid*. Channels are placed into various library states depending on the channel activity and resource allocation situations.

Event

No event results from the call to *irLibState*.

Return Value

A channel state [see *IrSTATES(4IRAPI)*] is returned if *irLibState* is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if the *cid* is invalid.

See Also

irServiceState(3IRAPI), IrSTATES(4IRAPI)

irMonitor

Name

irMonitor — Start/stop monitoring another channel

Synopsis

#include <**irapi.h**>

int irMonitor (channel_id *cid1***, int** *chan2***, int** *flag***);**

Description

The *irMonitor* function allows *cid1* to start/stop monitoring another channel (*chan2*). Monitoring means that all audio input and output on *chan2* is heard by *cid1* but *chan2* does not hear any audio input or output from *cid1*.

The value of *flag* determines if monitoring should be started or stopped. Use IRD_ADD [see *IrDEFINES(4IRAPI)*] as *flag* to start monitoring and IRD_DROP to stop monitoring.

Each channel can have at most 7 output timeslots, of which one is pre-assigned. Therefore, the channel can only add another 6 output timeslots.

Event

No event results from the call to *irMonitor*.

Return Value

IRR_OK is returned if *irMonitor* is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid1*, *chan2* or *flag* are not valid

IRER_MAXCHAN_TIMESLOTS if the channel has reached the limit for output timeslots (7 maximum per channel)

IRER_DRIVER_ERROR if a driver call failure occurs (see *irSysError* for additional information)

IRER_SYSERROR if a system or driver call failure occurs (see *irSysError* for additional information)

IRER_OVERFLOW if one channel attempts to monitor more than one other channel

IRER_REDUNDANT if the requested monitor is a duplicate request in the case where flag is IRD_ADD, or the requested monitor does not exist in the case where flag is IRD_DROP

See Also

irHBridge(3IRAPI)

irName

Name

irPName, irSName, irEName, irEMName, irAName, irSvcStName, irCName, irPrintEvent — Return symbolic names of various IRAPI constants

Synopsis

#include <**irapi.h**>

char *irPName (int *p***)**

char *irSName (int *s***)**

char *irEName (int *e***)**

char *irEMName (int *em***)**

char *irAName (int *a***)**

char *irSvcStName (int *ss***)**

char *irCName (int *c***)**

char *irPrintEvent (ir_event_t **evp***)**

Description

These functions return character pointers to the symbolic names of the values they are passed for the class of names implied by the function called.

irPName(3IRAPI) — Return a pointer to a parameter name [see *IrPARAMETERS(4IRAPI)*]

irSName(3IRAPI) — Return a pointer to a library state name [see *IrSTATES(4IRAPI)*]

irEName(4IRAPI) — Return a pointer to an event name [see *IrEVENTS(4IRAPI)*]

irEMName(3IRAPI) — Return a pointer to an event modifier name [see *IrEVENTS(4IRAPI)*]

irAName(3IRAPI) — Return a pointer to an algorithm name [see *IrALGORITHMS(4IRAPI)*]

irSvcStName(3IRAPI) — Return a pointer to a service state name [see *IrDEFINES(4IRAPI)*]

irCName(3IRAPI) — Return a pointer to a capability name [see *IrRESOURCES(4IRAPI)*]

irPrintEvent(3IRAPI) — Return a pointer to a formatted string containing all elements of an event structure [see *IrEVENTS(4IRAPI)*]

Event

No event results from the call to any of these functions.

Return

Character pointer to symbolic name of the IRAPI constant on success.

A character pointer to a string indicating that the value is unknown for the class implied by the function on error.

Warning

For unknown values, the return value points to a string that could possibly be overwritten by a call to the same or any other *irName* function listed in this man page.

Error

These functions always return a pointer to some valid address. *irError* is never modified as a result to a call to any of these functions.

See Also

irErrorStr(3IRAPI)

irNumChans

Name

irNumChans — Get the number of channels on the system

Synopsis

#include <**irapi.h**>

int irNumChans (int *type***);**

Description

The *irNumChans* function returns the number of channels of the given *type* on the system. *Type* may be either IRD_REAL, IRD_VIRTUAL, or the bitwise OR of the two values (IRD_REAL | IRD_VIRTUAL) [See *IrDEFINES(4IRAPI)*].

Event

No event results from the call to *irNumChans*.

Return Value

The number of channels, system wide, of the indicated type(s) is returned.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if *type* is invalid (not IRD_REAL, IRD_VIRTUAL or IRD_REAL | IRD_VIRTUAL).

irOpen

Name

irOpen — Open a voice file

Synopsis

#include <**irapi.h**>

vf_descriptor irOpen (const char *****voice_file*, int** *oflag***, int** *mode***);**

Description

The *irOpen* function opens a named *voice_file* for access by other IRAPI functions [such as *irPlay(3IRAPI)*, *irRecord(3IRAPI)*, and *irLSeek(3IRAPI)*]. The function returns voice file descriptor (*vfd*) that is used by these voice routines.

See *open(2)* for a description and values of *oflag* and *mode*. Note, however, that with *open(2)*, mode is optional whereas with *irOpen* it is required.

See *IrPARAMETERS(4IRAPI)* for a discussion of IRP_CREATE_MODE and IRP_CREATE_FLAG used when creating new files via *irRecord(3IRAPI)*.

Calls to *irPlay(3IRAPI)* or *irRecord(3IRAPI)* fail if the *vfd* has not been opened in a manner appropriate for those calls.

Event

No event results from the call to *irOpen*.

Return Value

This function returns a voice file descriptor (*vfd*) if successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if *voice_file* is a NULL pointer or string

IRER_SYSERROR if a system call failure occurs (see *irSysError* for additional information)

IRER_OVERFLOW if *voice_file* is greater than IRD_MAX_FILE_LEN

See Also

open(2), irClose(3IRAPI)

irParam

Name

irSetParam, irGetParam, irSetParamStr, irGetParamStr, irInitParam, irInitAllParams — Set/get/initialize the value of one or more IRAPI library channel-based parameters

Synopsis

#include <**irapi.h**>

int irGetParam (channel_id *cid***, int** *identifier***, int ***value***);**

int irGetParamStr (channel_id *cid***, int** *identifier***, char ***value***, int** *count***);**

int irSetParam (channel_id *cid***, int** *identifier***, int***value***);**

int irSetParamStr (channel_id *cid***, int** *identifier***, const char ***value***);**

int irInitParam (channel_id *cid***, int** *identifier***);**

int irInitAllParams (channel_id *cid***);**

Description

The *irSetParam* and *irSetParamStr* functions assign values to an IRAPI library parameter for later use on the specified channel *cid* while *irGetParam* gets the current value of a parameter. See *IrPARAMETERS(4IRAPI)* for a list of valid parameters and their default and legal values.

irGetParamStr copies, at most, *count* bytes into *value*. The application developer must verify that *value* is large enough; the number of bytes required can be found by calling *irGetParamStr* with *count* set to 0.

irGetParamStr returns exactly *count* bytes of data to the area specified by *value*. Since some *string* parameters are actually blocks of data, such as IRP_REGISTER, *irGetParamStr* ignores any NULL characters in the parameter data. It also makes no attempt to NULL terminate the *string*. *irSetParamStr* also ignores NULL characters and copies the number of bytes for the parameter [specified in *IrPARAMETERS(4IRAPI)*] into the call profile. Use *irGetParamStr* and *irSetParamStr* only with pointers to data objects at least as large as the size requirements specified in *IrPARAMETERS(4IRAPI)*.

irInitParam allows an application to set parameter *identifier* back to its default value for channel *cid*.

irInitAllParams allows an application to set all parameters for the *cid* to default values.

Parameters are preserved across *irExec(3IRAPI)* boundaries.

Example

The following example shows an application that wants to record speech. It uses *irGetParam and irSetParam* to first check current values of two recording parameters, sets the parameters if they are not appropriate, and then calls the recording function:

```
/* contrived example */
int value;
irGetParam (cid, IRP_RECORD_ALGO, &value);
if ( value != IRA_A_CS16 )
    irSetParam(cid, IRP_RECORD_ALGO, IRA_A_CS16 );
irGetParam (cid, IRP_RECORD_TONE, &value );
if ( value != IRD_ON )
    irSetParam(cid, IRP_RECORD_TONE, IRD_ON);
irFRecord (cid, tag, vfile, cnt);
```

Return Value

irGetParamStr returns the number of bytes copied to *value* if *count* is positive. If *count* is 0, the number of bytes that would have been copied to *value* is returned.

irGetParam, irSetParam, and *irSetParamStr* return IRR_OK if successful.

All functions return IRR_FAIL if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid, if the *identifier* is not valid, if *count* is negative or if *value* is not valid for *identifier*

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

See Also

IrPARAMETERS(4IRAPI), irGlobalParam(3IRAPI)

irPendingCid

Name

irChan2PendingCid, irGrp2PendingCid — Return cid pending on channel or group init

Synopsis

#include <**irapi.h**>

channel_id irChan2PendingCid (int *chan***)**

channel_id irGrp2PendingCid (int *group***)**

Description

The *irChan2PendingCid* function returns a *channel_id* pending on channel *chan*.

The *irGrp2PendingCid* function returns a *channel_id* pending on a channel in group *group*.

A pending *channel_id* results from a call to *irInit(3IRAPI)* or *irInitGroup(3IRAPI)* returning IRS_PENDING. While *irInit(3IRAPI)* and *irInitGroup(3IRAPI)* return the *channel_id* when called, functions *irChan2PendingCid* and *irGrp2PendingCid* are convenience functions for applications not wishing to maintain tables of *channel_id*s for outstanding channel requests.

Event

No event results from a call to either of these functions.

Return Value

The *channel_id* pending on the indicated channel or group if the request is successful.

IRD_NULL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *chan* is invalid

IRER_NOTFOUND if no *channel_id* is found pending on the indicated channel or group

See Also

irInit(3IRAPI), *irInitGroup(3IRAPI)*.

Caveats

Nothing prevents a single process from having multiple initialization requests for a single channel or group. Functions *irChan2PendingCid* and *irGrp2PendingCid* will simply return the first *channel_id* found matching the channel or group parameter.

irPhReserve

Name

irPhReserve — Reserve space for subsequent voice recording

Synopsis

#include <**irapi.h**>

int irPhReserve (channel_id *cid***, int** *tag***, int** *count***, const char *****file***);**

Description

The *irPhReserve* function reserves *count* milliseconds of space in the UNIX file system to record a file encoded in the algorithm defined in IRP_RECORD_ALGO. Space is reserved in *file*. Absolute or relative paths are allowed.

If *file* is NULL, a unique file name is generated. The directory is specified by the IRP_SPEECHDIR and IRP_TALKFILE parameters. The file name consists of all numeric characters. The API selects the first unique file name starting with "65535" and counts down until a unique name is found. The absolute file name is returned with the subsequent IRE_RESERVE_DONE event.

For example, if *irPhReserve* is called with *file* set to the NULL pointer, IRP_SPEECHDIR is set to */home2/vfs/talkfiles*, IRP_TALKFILE is set to *255* and there is no file */home2/vfs/talkfiles/255/65535*, */home2/vfs/talkfiles/255/65535* is the unique file name returned with the IRE_RESERVE_DONE event.

Event

The IRE_RESERVE_DONE event occurs with *event_mod1* set to IREM_COMPLETE or IREM_ERROR depending on the success or failure, respectively. *event_text* points to the file name reserved if successful. While the *cid* awaits this event, it is placed in the IRS_RESERVING library state.

Return Value

IRR_OK is returned if the request to reserve space is successfully initiated.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* or *count* are not valid.

IRER_OVERFLOW if the file name specified by *file* is greater than IRD_MAX_FILE_LEN characters in length. This error may also be produced as a result of converting a relative file name into an absolute file name.

IRER_SYSERROR if a system or driver call failure occurs (check *irSysErrror* for additional information)

irPlay

Name

irPlay, irFPlay, irBPlay — Play speech from a voice file descriptor, a voice file name, or a buffer

Synopsis

#include <**irapi.h**>

int irPlay (channel_id *cid***, int** *tag***, vf_descriptor** *vfd***, unsigned long** *count***);**

int irFPlay (channel_id *cid***, int** *tag***, char *** *voice_file***);**

int irBPlay (channel_id *cid***, int** *tag***, char *** *buf***, unsigned long** *count***);**

Description

These functions play speech on a voice channel indicated by a channel identifier (*cid*). The *cid* is obtained by calling *irInit(3IRAPI)*.

Tag is a user-supplied number that associates a play function call with a subsequent event.

irPlay plays *count* milliseconds of speech from the file associated with an open voice file descriptor *vfd*. The *vfd* is obtained from *irOpen(3IRAPI)*, and discarded by *irClose(3IRAPI)*. Play starts from the current position of the voice-pointer. The voice pointer is initially set to 0 when the voice file is opened [*irOpen(3IRAPI)*]. However, the voice pointer may be repositioned by calling *irLSeek(3IRAPI)*. *irPlay* does not modify the voice pointer for the *vfd* it is using. If *count* is 0, then play continues until the end of file.

irFPlay plays speech from a voice file identified by the file name *voice_file*. *irFPlay* opens the *voice_file*, plays the entire file from the beginning, then automatically closes the file.

irBPlay plays *count* bytes of speech from a voice buffer identified by the buffer name *buf*. The data represented by *buf* and *count* cannot be reused until the play is complete (that is, IRE_PLAY_DONE occurs).

The IRAPI library parameter IRP_PLAY_GAIN specifies the play volume and IRP_PLAY_SPEED specifies play speed. Existing values for these parameters can be retrieved using *irGetParam(3IRAPI)* while new values can be assigned to them using *irSetParam(3IRAPI)*. The default IRP_PLAY_GAIN is unity gain (0 db) and IRP_PLAY_SPEED is unity. These parameters must be set to the desired values before invoking a play function. Play functions use the value of these parameters available at the time of the play.

Any play function must be followed by *irEnd(3IRAPI)* and can be terminated when an enabled interrupt occurs [see *irSetEvent(3IRAPI)*]. When *irFPlay* is terminated, the *voice_file* is closed.

If the library state is IRS_IDLE (not playing), calling a play function changes the library state to IRS_PLAY_QUEUED. If the library state is IRS_PLAY_QUEUED, subsequent calls to a play function are queued and the library remains in the IRS_PLAY_QUEUED state. See *irIntro(3IRAPI)* for a discussion of library states.

Invocation of multiple play functions to a single *cid* results in concatenated speech to that channel. The speech stream, up to a "terminating" *irEnd(3IRAPI)*, are played seamlessly. Any of the play functions may be intermixed before the list is terminated by *irEnd(3IRAPI)*.

irGetVCount(3IRAPI) is used to obtain the number of bytes played after any of the play functions are interrupted or stop playing. *irGetVCount* only provides meaningful results after *irEnd(3IRAPI)* and one or more IRE_PLAY_PROG events have occurred (if IRE_PLAY_PROG is enabled). If IRE_PLAY_PROG is not enabled, *irGetVCount(3IRAPI)* returns meaningful results until IRE_PLAY_DONE occurs.

Event

An IRE_PLAY_DONE event is generated when all play requests preceding the *irEnd(3IRAPI)* complete or an interrupt occurs which causes play to stop. If IRE_PLAY_PROG is enabled, upon completion of all but the last play requests queued before the *irEnd(3IRAPI)*, an IRE_PLAY_PROG event occurs. Upon completion of the last play event, an IRE_PLAY_DONE event occurs.

Return Value

All play functions return IRR_OK if the play request is queued successfully.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid from all functions

IRER_INVALID if the *vfd* is invalid from *irPlay*

IRER_INVALID if the *buf* is the NULL pointer or if *count* is less than or equal to zero

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_BADSTATE if the *cid* is not in the IRS_IDLE or IRS_PLAY_QUEUED state

IRER_OVERFLOW if the *voice_file* is greater than IRD_MAX_FILE_LEN

irFPlay does not check for the existence of *voice_file*. Failure to find a file specified through *irFPlay* results in a logger message generated by VROP and an IREM_ERROR modifier on the subsequent IRE_PLAY_DONE event.

Caveat

Playing a speech file descriptor, a file, or a buffer of mixed coding algorithms has unpredictable results.

See Also

irSay(3IRAPI), IrPARAMETERS(4IRAPI)

irPlayKill

Name

irPlayKill — Stop play immediately

Synopsis

#include <**irapi.h**>

int irPlayKill (channel_id *cid***);**

Description

The *irPlayKill* function immediately stops a play activity on the channel and forces the channel into the IRS_IDLE state.

irGetVCount(3IRAPI) does not contain valid results after a play is terminated with *irPlayKill*.

Event

No event results from a call to this function.

Return Value

IRR_OK is returned if the request is successful. IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_BADSTATE if the channel is not in the IRS_PLAYING state

IRER_SYSERROR if a driver call or system call failure occurs (check *errno* for additional information)

See Also

irPlay(3IRAPI), *IrSTATES(4IRAPI)*.

Caveat

This function is provided to support old voice system behavior and is not intended for use by well designed IRAPI applications. Since this function immediately forces the library into the idle state, despite the fact that voice output may still be active for a short period of time, undesirable side effects may be produced on the channel such as mixing of voice and touch-tone output. Multitasking on some telephony cards may produce unrecoverable errors.

irPlayResume

Name

irPlayResume — Resume voice play

Synopsis

#include <**irapi.h**>

int irPlayResume (channel_id *cid***, int** *offset***);**

Description

The *irPlayResume* function may be used to restart a *remembered* play or record request from the point it was interrupted, stopped or completed, relative to *offset*. *offset* is specified in milliseconds. See *irEnd(3IRAPI)* or *irRecord(3IRAPI)* for information on how to remember play and record requests.

irPlayResume may only be called from the IRS_IDLE state.

Event

The IRE_PLAY_DONE event occurs when play completes. The IRE_PLAY_PROG event only occurs if it was enabled when the remembered play was initially specified.

Return Value

IRR_OK is returned if the request to resume play is successfully initiated.

IRR_PENDING is returned if resources could not be allocated and IRP_RESOURCE_RETURNMODE is not set to IRD_IMMEDIATE.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid or if *cid* has not remembered any play or record request. Note that buffer record requests cannot be remembered and therefore can not be play resumed.

IRER_BADSTATE if the *cid* is in any state other than IRS_IDLE

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_RESOURCEBUSY if there are no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE

IRER_NORESOURCES if there are no resources in existence

IRER_RESTRICTED if the channel is restricted from using the required resources

See Also

irEnd(3IRAPI) and *irPlay(3IRAPI)*

irPostEvent

Name

irPostEventC, irPostEventQ, irPostEvent — Post an event to an application or channel

Synopsis

#include <**irapi.h**>

int irPostEventC (int *channel_nbr***, const void ***mbuf**, int** *len***);**

int irPostEventQ (ir_key_t *q_key***, const void ***mbuf**, int** *len***);**

int irPostEvent (const void ***mbuf, int** *len***);**

Description

The *irPostEventC* and *irPostEventQ* functions allow processes to communicate with each other by generating events on the indicated *channel_nbr* or *q_key*. *mbuf* is a pointer to a message buffer to be sent to the target channel or queue.

mbuf must be a user allocated structure of the following form:

```
struct msgBuf {
    struct mbhdr header;
    char udata[VARSIZE];
} msgBuf;

struct msgBuf *mbuf;
```

struct mbhdr is defined in *mesg.h*
(which is included by *irapi.h*) as follows:

```
struct mbhdr {
    long mtype;
    short irType;
    short irWhoTo;
    long irChan;
    long mchan;
    short morig;
    short mcont;
    unsigned short mseqno;
};
```

Applications using *irPostEventQ* and *irPostEventC* are not required to set any of the elements in the *mbhdr* structure. *irWhoTo*, *irChan*, *irType* and *morig* are always set by IRAPI with these functions. Applications may use the remaining fields for their own purposes.

irPostEvent requires that the application set *irWhoTo* and *irChan* directly. *irPostEvent* sets *irType* and *morig* internally. Applications may use the remaining fields for their own purposes. Applications should set *irWhoTo* to the Qkey of the destination or to IRD_IRAPI if the event is channel based. *irChan* should be set to the channel number if the message is intended for a channel or IRD_INVALID if the message is intended for a queue.

The IRE_EXTERNAL event is generated on the target channel or queue. See *IrEVENTS(4IRAPI)* for a description of the event structure and possible events. The event structure element *event_text* points to a copy of the *mbuf*. The receiving process should copy the contents of *event_text* to a private buffer if needed between calls to *irWait(3IRAPI)* since its contents are not preserved.

The event structure sets the *cid* to the channel_id of the receiving channel if the event is generated via *irPostEventC* or is channel based. Otherwise, *cid* is set to IRD_INVALID.

Event

The IRAPI library does not return any event to the application as a result of calling *irPostEventC*, *irPostEventQ* or *irPostEvent*.

An IRE_EXTERNAL event occurs for the receiving process owning the channel or the queue.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_INVALID if either *channel_nbr* or *q_key* is invalid or if the *mbhdr* structure elements *irWhoTo* or *irChan* are invalid

Example

These functions can be used by a pair of applications to perform asynchronous operations. Process A is a multi-threaded IRAPI application. Process B is a database server process. Process A may, on behalf of some caller, request a database transaction via *irPostEventQ* to the process B message queue. Process A is now free to service other channel's needs and eventually calls *irWait(3IRAPI)*. The details of the database transaction may be contained in the message text *(struct msgBuf *)(event.event_text)->udata* . Process B, waiting in *irWait(3IRAPI)*, receives an IRE_EXTERNAL event. The event is processed and the results of the database transaction are transmitted to process A via some *irPostEvent* function.

Processes A and B identify their queue keys when *irRegister(3IRAPI)* is called. Process A identifies the queue key of process B via *irGetQKey(3IRAPI)*. Process B identifies process A's queue key when it receives an IRE_EXTERNAL event from process A and examines the *morig* element of the message buffer header.

See Also

irRegister(3IRAPI), *irGetQKey(3IRAPI)* and *IrEVENTS(4IRAPI)*.

irQueryResource

Name

irQueryResource — Return a list of resources on the system

Synopsis

#include <**irapi.h**>

int irQueryResource (int *list_size***, int** *capability***, int** *implementation***, ir_resource_t *****pir_resource***);**

Description

The *irQueryResource* function returns a list of the resources with *capability* and *implementation* currently in service. The size of the list is equal to the smaller of *list_size* or the actual number of resources matching the query. If *capability* is IRC_NULL, all capabilities and implementations are returned. If *implementation* is IRD_INVALID, all implementations are returned. If *capability* is IRC_NULL, the value of *implementation* has no effect on the results of the query.

The resource capabilities and implementations are defined in *IrRESOURCES(4IRAPI)*.

List_size is the number of elements expected in the returned list and *pir_resource* is a pointer to an array of *ir_resource_t* elements. The results of the query are placed in this array. See *IrRESOURCES(4IRAPI)*.

The *ir_resource_t* structure contains a resource capability field and an array of resource numbers providing that capability. The resource numbers are the hardware components which support the capabilities. In most cases, it is the board numbers providing the resources for the capability. Note that more than one hardware component may provide resources for some capability, so more than one resource number may be provided per capability. Also note that a single hardware component may support multiple capabilities such as a single SP/CMP card pair supporting IRC_RECOG, IRC_ECHOCAN, IRC_RECORD and IRC_PLAY.

If *pir_resource* is NULL, *irQueryResource* returns the total number of resources/implementation pairs matching the query. This behavior can be applied as follows:

```
int j = irQueryResource ( 0, IRC_NULL, IRD_INVALID,  NULL);
ir_resource_t *list =
    calloc ( j, sizeof(ir_resource_t));
irQueryResource ( j, IRC_NULL, IRD_INVALID, list );
```

Event

No event results from the call to *irQueryResource*.

Return Value

The *irQueryResource* function returns the number of resources in matching the query if successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if *list_size* is less than zero and *pir_resource* is non-NULL

IRER_SYSERROR if a driver call or system call failure occurs (check *irSysError* for additional information)

irRecog

Name

irStartRecog, irStopRecog, irCheckRecog — Start, stop, check state of speech recognizer for channel

Synopsis

#include <**irapi.h**>

int irStartRecog (channel_id *cid***, int** *tag***);**

int irStopRecog (channel_id *cid***);**

int irCheckRecog (channel_id *cid***);**

Description

The *irStartRecog* function starts speech recognition for *cid*. Parameters IRP_RECOG_TYPE and IRP_RECOG_GRAMMAR, set and accessed via *irSetParam* and *irGetParam*, determine the type of recognition and the specific grammar for the recognition type. See *IrPARAMETERS(4IRAPI)*. IRP_RECOG_TYPE may be either IRD_WHOLE_WORD or IRD_FLEX_WORD. IRP_RECOG_GRAMMAR depends on the recognition type.

tag is returned to the application through the subsequent IRE_GRANT or IRE_DENY events, on delayed resource allocation, or through the IRE_INPUT_DONE event associated with the recognition.

Note that recognition uses the IRE_INPUT_DONE event rather than the IRE_INPUT event since speech recognizers do not provide continuous input as touch-tone recognizers do. Once the speech recognizer provides input, it is finished and must be restarted for subsequent recognitions.

Event

If resources are not available when *irStartRecog* is called and IRP_RESOURCE_RETURNMODE is *not* set to IRD_IMMEDIATE, recognition does not start until the IRE_GRANT event occurs. IRE_DENY may occur if resources are not available within the time limit specified in IRP_RESOURCE_RETURNMODE. See *irIntro(3IRAPI)* for a general discussion of the resource allocation strategy.

When recognition completes, an IRE_INPUT_DONE event is generated. See *irRecogTimer(3IRAPI)*. The recognizer is automatically turned off after receiving the IRE_INPUT_DONE event.

No event results from the call to *irStopRecog* or *irCheckRecog*.

Return Value

For *irStartRecog*, the following return codes are possible:

IRR_OK if the request is successful

IRR_PENDING if all recognition resources are currently occupied. See the EVENT section above for handling of this condition.

IRR_FAIL if an error occurs

For *irStopRecog*, the following return codes are possible:

IRR_OK if the request to stop speech recognition is successful

IRR_FAIL if an error occurs

For *irCheckRecog*, the following return codes are possible:

IRR_ON if recognition is on

IRR_OFF if recognition is off

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_RESOURCEBUSY if there are no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE

IRER_NORESOURCES if resources do not exist

IRER_RESTRICTED if the channel is restricted from using the required resources

IRER_DRIVER_ERROR if a driver call failure occurs (check *irSysError* for additional information)

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_BADSTATE if the *cid* is not in the IRS_IDLE, IRS_PLAY_QUEUED or IRS_SAY_QUEUED library state

See Also

irEcho(3IRAPI), IrDEFINES(4IRAPI), IrEVENTS(4IRAPI)

irRecogTimer

Name

irStartRecogTimer — Start the recognition input string timer

Synopsis

#include <**irapi.h**>

int irStartRecogTimer (channel_id *cid***);**

Description

The *irStartRecogTimer* function starts the timer used for a recognition input string collection based on timer expiration.

IRE_INPUT_DONE event [see *IrEVENTS(4IRAPI)*] occurs when certain conditions are satisfied. For instance, this event may be generated if the recognition input timer expires. The time interval starts when *irStartRecogTimer* is called. The IRAPI library parameter IRP_RECOG_PRETIME [see *IrPARAMETERS(4IRAPI)*] specifies the time period to wait before the initial digit is spoken.

irStartRecogTimer starts a timer for IRP_RECOG_PRETIME milliseconds. If an utterance is detected before the timeout occurs, the recognition timer is canceled. The IRAPI automatically manages clock timeouts from the point when *irStartRecogTimer* is called until IRE_INPUT_DONE is generated. Note: IRP_RECOG_PRETIME must be set to the desired value when *irStartRecog(3IRAPI)* is executed. *irStartRecogTimer* simply informs the recognizer to start timing with the value previously supplied when the recognizer was started via *irStartRecog(3IRAPI)*.

If a recognition timeout occurs before an utterance is recognized, an IRE_INPUT_DONE event is generated with the IREM_RECOG_PRE modifier.

The existing value of the IRP_RECOG_PRETIME parameter can be retrieved using *irGetParam(3IRAPI)* while a new value can be assigned to a parameter using *irSetParam(3IRAPI)*. Defaults for each are 5000 milliseconds.

These parameters must be set to the desired value before invoking *irStartRecogTimer*.

irStartRecogTimer may only be called after the recognizer has been turned on via *irStartRecog(3IRAPI)*.

Other conditions under which IRE_INPUT_DONE event is generated are explained in *irGetInput(3IRAPI)*.

Calling *irStartRecogTimer* with an outstanding recognition active cancels the outstanding timer and starts a new timer as defined above.

Events

An IRE_INPUT_DONE event is generated after *irStartRecogTimer* is called if the conditions explained above are satisfied.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_BADSTATE if recognition has not been started

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information).

See Also

IrPARAMETERS(4IRAPI), IrEVENTS(4IRAPI), irSetParam(3IRAPI), irGetParam(3IRAPI), irGetInput(3IRAPI), irStartRecog(3IRAPI)

irRecord

Name

irRecord, irFRecord, irBRecord — Record speech into a voice file via a voice file descriptor, into a voice file via a voice file name, or into a voice buffer

Synopsis

#include <**irapi.h**>

int irRecord (channel_id *cid***, int** *tag***, vf_descriptor** *vfd***, long** *count***, int** *remember***);**

int irFRecord (channel_id *cid***, int** *tag***, const char *** *voice_file***, long** *count***, int** *remember***);**

int irBRecord (channel_id *cid***, int** *tag***, char *** *buf***, long** *count***);**

Description

These functions record speech over a voice channel indicated by a channel identifier (*cid*). The *cid* is obtained by calling *irInit(3IRAPI)*.

Tag is a user supplied number that associates the record function call with a subsequent event. The event is returned in an event structure (*ir_event_struct*) that is accessed by *irCheck(3IRAPI)*.

The *remember* value, when set to IRD_TRUE, indicates that this record request should be remembered for subsequent recording or playing via *irRecordResume(3IRAPI)* or *irPlayResume(3IRAPI)*.

irRecord records *count* milliseconds of speech into a voice file associated with an open voice file descriptor (*vfd*). *vfd* is obtained by *irOpen(3IRAPI)*, and discarded by *irClose(3IRAPI)*. Recording starts from the current position of the voice pointer. The voice pointer is initially set to 0 when the voice file is opened (*irOpen(3IRAPI)*). However, the voice pointer may be repositioned by calling *irLSeek(3IRAPI)*. *irRecord* overwrites the file from the position of the voice pointer. If positioned at the end of a voice file, *irRecord* appends to the existing voice data. If *count* is 0, then recording continues until a system hardware limit is reached or an interrupt occurs. If a file is opened in O_APPEND mode, the voice pointer is set to the end of the file.

irFRecord(3IRAPI) opens the named *voice_file*, records the *voice_file* from the beginning, then closes the file. Up to *count* milliseconds are recorded unless interrupted (see discussion of interrupt below). If *count* equals 0, then *irFRecord* records until terminated by silence timeout, another interrupt, or a write error.

Voice_file is created with a default "0644" permission mode, unless IRP_CREATE_MODE specified some other value [see *IrPARAMETERS(4IRAPI)*].

irBRecord records *count* milliseconds of speech by copying IRD_SPEECH_BUF_SIZE bytes of data to *buf* every time the IRE_RECORD_BUF event is generated. The application developer must allocate IRD_SPEECH_BUF_SIZE bytes of data space to which *buf* points. For every IRE_RECORD_BUF event, the application should complete processing of the data pointed to by *buf* before calling *irWait(3IRAPI)* or *irWCheck(3IRAPI)*. This data is overwritten as each IRE_RECORD_BUF event is generated.

The IRP_RECORD_GAIN parameter specifies recording volume gain. The gain is an adjustment to the default channel gain specified through system administration (refer to Chapter 6, "Switch Interface Administration," of *Intuity CONVERSANT VIS V5.0 Operations*, 585-310-550). The IRP_RECORD_TONE parameter specifies whether a recording tone is to be used and the IRP_RECORD_ALGO parameter specifies the recording algorithm to be used by any record function. IRP_RECORD_PRETIME specifies the initial silence timeout and IRP_RECORD_INTERTIME specifies the interutterance timeout. All parameters are set to reasonable defaults [see *IrPARAMETERS(4IRAPI)*]. Existing values for these parameters can be retrieved using *irGetParam(3IRAPI)* while new values can be assigned to them using *irSetParam(3IRAPI)*.

These parameters must be set to the desired values before invoking a record function. Record functions use the value of these parameters available at the time the function is called.

Any record function can be terminated when an enabled interrupt occurs [see *irSetEvent(3IRAPI)*]. An IRE_RECORD_DONE event is generated when any record function completes recording. If a silence event (IREM_RECORD_DONE with the IREM_SILENCE *event_mod1* modifier) occurs *irBRecord(3IRAPI)* does not truncate any silence. The application developer must delete the silence from the current buffer.

If the library state is IRS_IDLE (not recording), calling any record function changes the library state to IRS_RECORDING or IRS_RECORD_PENDING if record resources are not available. See *irIntro(3IRAPI)* for a discussion of library states.

irGetVCount(3IRAPI) is used to obtain the number of bytes recorded after any of the record function is interrupted or stops recording.

Event

An IRE_RECORD_DONE event is generated when any record function completes. The reason for record completion is contained in *event_mod1*.

An IRE_RECORD_BUF is generated only when *irBRecord* is used. This event, as detailed above, indicates that the user buffer has been populated with the latest recorded data.

Return Value

All record functions return IRR_OK if the record request is successfully initiated.

IRR_FAIL is returned if an error occurs.

IRR_PENDING is returned if code resources are not available. See *irIntro(3IRAPI)* for what conditions may lead to a return code of IRR_PENDING.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid or if the *vfd* is not valid for *irRecord*

IRER_BADSTATE if the channel is not in the IRS_IDLE state

IRER_RESOURCEBUSY if there are no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE

IRER_NORESOURCES if no resources exist

IRER_RESTRICTED if the channel is restricted from using the required resources

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

See Also

irPlay(3IRAPI), *irByte2Time(3IRAPI)*, *irTime2Byte(3IRAPI)*, *irOpen(3IRAPI)*, *irRecordResume(3IRAPI)*, *irPlayResume(3IRAPI)*, *irGetVCount(3IRAPI)* and *IrEVENTS(4IRAPI)*.

Caveat

Be aware that *irBRecord* requires approximately twice the CPU resources of a record done via *irRecord* or *irFRecord*. It is also important that applications making use of *irBRecord* minimize the amount of time spent processing the recorded speech before IRE_RECORD_DONE is generated. Delays due to non-IRAPI blocking function calls, etc., may result in an overflow of available speech buffers.

irRecordResume

Name

irRecordResume — Resume voice record

Synopsis

#include <**irapi.h**>

int irRecordResume (channel_id *cid***, int** *offset***);**

Description

The *irRecordResume* function may be used to restart a *remembered* play or record request from the point it was interrupted, relative to *offset*. See *irEnd(3IRAPI)* and *irRecord(3IRAPI)* for information on how to remember a play or record request. *offset* is specified in milliseconds.

irRecordResume may only be called from the IRS_IDLE state. Also, *irRecordResume* may only be called on a remembered *play* request if the play request included a single voice file or voice file descriptor; it is not possible to record over a remembered multi-voice file or buffer play request.

Event

The IRE_RECORD_DONE event occurs when record completes.

Return Value

IRR_OK is returned if the request to resume record is successfully initiated.

IRR_PENDING is returned if resources could not be allocated and IRP_RESOURCE_RETURNMODE is not set to IRD_IMMEDIATE.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid or if the *cid* has not remembered any play or record request. IRER_INVALID is also set if *irRecordResume(3IRAPI)* is called on a remembered *play* request containing multiple voice file or voice file descriptors or both or if the remembered play activity contains a buffer play.

IRER_BADSTATE if the *cid* is in any state other than IRS_IDLE

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_RESOURCEBUSY if no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE

IRER_NORESOURCES if no resources in exist.

IRER_RESTRICTED if the channel is restricted from using the required resources

See Also

irEnd(3IRAPI) and *irRecord(3IRAPI)*

irRegister

Name

irRegister — Register a process with the voice system

Synopsis

#include <**irapi.h**>

ir_key_t irRegister (const char * *name*);

Description

The *irRegister* function registers the calling process, referred to as *name* in an IRAPI application. Once registered, this process may now receive events generated on its behalf.

Many of the functions in the IRAPI library only work properly after the application process has called *irRegister*, so it must be called early during the initialization. Processes that use the IRAPI library without calling *irRegister* may encounter undesired results.

Event

No event is generated as a result of a call to *irRegister*.

Return Value

The QKey of the calling process is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if *name* is not valid, exceeds IRD_MAX_APP_NAME in length, or is already in use

IRER_DRIVER_ERROR if a driver call failure occurs (check *irSysError* for additional information)

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

See Also

*irPostEvent(3IRAPI)*and *irGetQKey(3IRAPI)*.

irReserveResource

Name

irReserveResource — Reserve resources for later use

Synopsis

#include <**irapi.h**>

int irReserveResource (channel_id *cid***, int** *tag***, ir_reserve_t *****presources***, long** *mode***, int *****failure***);**

Description

The *irReserveResource* function acquires or allocates one or more resources for later use on a particular voice channel (*cid*). Only one resource of each type (capability/implementation) may be reserved per channel.

Normally, a process automatically allocates a single resource by using it. For example, playing a voice file uses (allocates) the resources necessary to play the file and performing speech recognition allocates ASR resources. A resource then is deallocated automatically after using it. *irReserveResource* allows an application to insure that a resource is available **before** using it. Typically this should be done immediately after channel initialization [*irInit(3IRAPI)*]. The application frees the resource when it is no longer required by calling *irFreeResource(3IRAPI)*.

If *irReserveResource* succeeds, the requested resources are dedicated to the channel. Other channels can not access this channel's allocated resource until either the owner relinquishes control [via *irFreeResource(3IRAPI)*], the channel is released via *irDeinit(3IRAPI)*, or the channel owner *exit(2)*s. Since this dedication can cause requests for resources from other channels to fail that normally would succeed, *irReserveResource* should be used with care. In general, an application that uses *irReserveResource* uses more resources than an application that does not.

tag is a user supplied number that associates the *irReserveResource* call with a subsequent event.

presources is a pointer to an array of capability/implementation pairs to be allocated. If implementation is set to IRD_INVALID, the implementation specific switch parameter [see *IrRESOURCE(4IRAPI)*] is referenced to determine the implementation.

If the requested resources are available, *irReserveResource* returns IRR_OK. *mode* defines the return operation of *irReserveResource* when the requested resource is not immediately available:

IRD_IMMEDIATE	If the requested resources are not available, *irReserveResource* returns IRR_FAIL.
IRD_BLOCKFOREVER	If the requested resources are not available, *irReserveResource* returns IRR_PENDING. The application then should wait for the IRE_GRANT event for this request assuming the resources are available.
+N	If the requested resources are not available, *irReserveResource* returns IRR_PENDING. The application waits for a maximum of "+N" milliseconds, or until the requested resources are available. IRE_DENY occurs if the resources are not available before timeout. IRE_GRANT occurs if the resources are granted within the timeout.

Once *irReserveResource* is called, no other processing can be done on the *cid* until the request for reserve is complete as the channel is placed in the IRS_RESERVING library state.

failure contains an index into the reserve array indicating the element which could not be allocated.

The reserved resources are preserved across *irExec(3IRAPI)* boundaries.

Event

The IRE_GRANT event is generated when *irReserveResource* succeeds after IRR_PENDING was returned. The IRE_DENY event is generated in the resources do not become available within +N milliseconds.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs. On a multi-resource request, all successful intermediate resource allocation made by the current call to *irReserveResource* before failure is freed before *irReserveResource* returns. That is, failure implies complete failure.

IRR_PENDING is returned if the request for the resource(s) is pending.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid*, *mode* or some capability in *presources* is not valid

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_RESOURCEBUSY if there are no resources available and IRP_RESOURCE_RETURNMODE is set to IRD_IMMEDIATE

IRER_NORESOURCES if no resources exist with the indicated capability

IRER_RESTRICTED if the channel is restricted from using the required resources

IRER_BADSTATE if the *cid* is not in the IRS_IDLE, IRS_PLAY_QUEUED, or IRS_SAY_QUEUED state

Caveat

Deadlock is possible if IRP_RESOURCE_RETURNMODE is set to IRD_BLOCKFOREVER.

Use of this function is not mandatory. It is provided for special cases where it is desirable to reserve resources in advance. Otherwise, all IRAPI functions implicitly reserve resources when needed.

See Also

irFreeResource(3IRAPI), IrRESOURCES(4IRAPI)

irRestrictResource

Name

irRestrictResource — Restrict a channel to a set of resources

Synopsis

#include <**irapi.h**>

int irRestrictResource (channel_id *cid***, ir_reserve_t *** *presources***);**

Description

The *irRestrictResource* function restricts a channel to a list of resources specified with *presources*. *presources* is an array of capabilities/implementation pairs terminated with IRC_NULL. Whereas the channel specified by *cid* is restricted to these resources, other channels may use those same resources. Different channels may be specified on the same or overlapping list of resources.

See *IrRESOURCES(4IRAPI)* for a definition of *ir_resource_t*.

The resource restriction list is preserved across *irExec(3IRAPI)* boundaries. The resource restriction list is removed on *irDeinit(3IRAPI)*.

Event

No event is generated as a result of the call to *irRestrictResource*.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs.

IRER_INVALID if the *cid* or some capability in *presources* is not valid

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

See Also

irQueryResource(3IRAPI), *irReserveRes(3IRAPI)*, *irFreeResource(3IRAPI)* and *IrRESOURCES(4IRAPI)*.

irSay

Name

irSay, irFSay, irBSay — Say ASCII text using text-to-speech from a file via file descriptor, from a file via file name, or from a buffer

Synopsis

#include <**irapi.h**>

int irSay (channel_id *cid***, int** *tag***, int** *fd***, unsigned long** *count***);**

int irFSay (channel_id *cid***, int** *tag***, const char *** *text_file***);**

int irBSay (channel_id *cid***, int** *tag***, const char *** *buf***, unsigned long** *count***);**

Description

The text-to-speech (TTS) functions say ASCII text on a voice channel indicated by a channel identifier (*cid*). These functions internally use TTS to convert ASCII text into speech. The *cid* is obtained by calling *irInit(3IRAPI)*.

tag is provided only for consistency with other functions and for backward compatibility with future extensions of the library which may require it. Application developers requiring a tag for "say" instructions may use the *tag* argument provided with *irEnd(3IRAPI)*.

The *irSay* function says *count* bytes of ASCII text from the file associated with an open UNIX file descriptor *fd*. *Fd* is obtained by *open(2)* and discarded by *close(2)*. "Saying" starts from the current position of the file pointer. The file pointer is initially set to 0 when the text file is opened (*open(2)*). However, the file pointer may be repositioned by calling *lseek(2)*. *irSay* does not modify the file pointer for the file descriptor it is using. If *count* is 0, then say continues until the end of file.

irFSay says ASCII text from a file identified by the file name *text_file*. *irFSay* opens the *text_file*, says the entire file from the beginning, then automatically closes the file.

irBSay says ASCII text from a buffer identified by the buffer name *buf*. The data represented by *buf* and *count* may be reused since *irBSay* copies the buffer to a private area. NULL characters are not interpreted as string terminators by *irBSay* as exactly *count* bytes of data are taken as input.

TTS functions interoperate with echo cancellation and barge-in.

Any TTS function must be followed by *irEnd(3IRAPI)*. Any TTS function can be terminated when an enabled or non-maskable interrupt occurs [see *irSetEvent(3IRAPI)*]. When *irFSay* is terminated the *text_file* is closed.

If the library state is IRS_IDLE (not saying), calling a TTS function changes the library state to IRS_SAY_QUEUED. If the library state is IRS_SAY_QUEUED, subsequent calls to a TTS function are queued and the library remains in the IRS_SAY_QUEUED state. The TTS functions may be called only when the library is in the IRS_IDLE or IRS_SAY_QUEUED states. See *irIntro(3IRAPI)* for a discussion of library states.

Invocation of multiple TTS functions to a single *cid* results in concatenated speech to be said on that channel. The speech stream, up to a "terminating" *irEnd(3IRAPI)*, said seamlessly. Any of the TTS functions may be intermixed before the list is terminated by *irEnd(3IRAPI)*. Upon calling *irEnd(3IRAPI)*, the library enters the IRS_SAYING or IRS_SAY_PENDING states depending on TTS resource availability.

Event

An IRE_SAY_DONE event is generated after *irEnd(3IRAPI)* is called and all queued *irSay* requests have completed or are interrupted [as indicated in the modifier to IRE_SAY_DONE, see *IrEVENTS(4IRAPI)*].

Return Value

IRR_OK is returned if the say request is successfully queued.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* or the *fd* are not valid

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_BADSTATE if the *cid* is not in the IRS_SAY_QUEUED or IRS_IDLE state

Note, for *irFSay*, if *text_file* does not exist, *irFSay* fails. This differs from *irFPlay(3IRAPI)* where non-existence of a voice file is indicated through modifiers on the IRE_PLAY_DONE event.

See Also

irPlay(3IRAPI), IrPARAMETERS(4IRAPI), IrEVENTS(4IRAPI)

irServiceState

Name

irServiceState — Return the service state of a channel

Synopsis

#include <**irapi.h**>

int irServiceState (channel_id *cid***);**

Description

The *irServiceState* function returns the service state identifier associated with the channel identifier (*cid*). The two most likely service states are IRD_ACTIVE and IRD_INACTIVE. These are analogous to a channel being off-hook or on-hook, respectively. Additional service states are described below and in *IrDEFINES(4IRAPI)*.

In general applications should react to reported events without needing to check the service state. Knowing the service state is useful when investigating IRER_SERVICE_STATE errors. That information often can be obtained by tracing the application that encounters the error.

The currently defined service states may be changed in future releases and additional service states may be provided by add on packages.

Some service states currently apply to all telephony types. IRD_INACTIVE means that a channel is idle or on-hook and available for use. IRD_RINGING means that a channel has detected an incoming call and is ready for the channel to be answered with *irAnswer(3IRAPI)*. IRD_ACTIVE means that the channel is off-hook or active. IRD_CHAN_OOS means that the channel is out of service [owned by maintenance (MTC)].

Some service states are specific to the primary rate interface (PRI). IRD_WAIT_CLEAR indicates that the IRAPI is waiting for the network to complete the clearing of the channel. IRD_REMOTE_DISCON indicates that the remote end has disconnected. IRD_REMOTE_CLEAR indicates that the call has been cleared from the network perspective, but the library is waiting for the IRAPI application to disconnect. IRD_WAIT_ALERT indicates that an incoming call is coming in, but the IRAPI is waiting for further exchange of information with the switch.

IRD_WAIT_DNIS is specific to Adjunct/Switch Application Interface (ASAI) channels and indicates that the IRAPI is waiting for the dialed number identification service (DNIS) and automatic number identification (ANI) information to be provided via the basic rate interface (BRI) D channel.

Event

No event is generated as a result of the call to *irServiceState*.

Return Value

irServiceState returns a service state, such as IRD_INACTIVE or IRD_ACTIVE, if successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if the *cid* is invalid.

See Also

irLibState(3IRAPI), *irName(3IRAPI)*.

irSpeechED

Name

irStartSpeechED, irStopSpeechED, irCheckSpeechED — Start, stop, or check the state of speech energy detection for a channel

Synopsis

#include <**irapi.h**>

int irStartSpeechED (channel_id *cid***);**

int irStopSpeechED (channel_id *cid***);**

int irCheckSpeechED (channel_id *cid***);**

Description

The *irStartSpeechED* function starts speech energy detection for channel identifier (*cid*). The *irStopSpeechED* function stops speech energy detection for *cid*. The *irCheckSpeechED* function checks the status of the speech energy detector.

irStartSpeechED currently is supported only for Tip/Ring (T/R) cards and can only be used while there is no other activity being performed by the T/R driver on that channel (it cannot be started while playing of speech is in progress or while a flash is being performed). The detection algorithm is intended primarily for detecting speech energy associated with someone answering the phone and may fail to detect short or quiet responses.

Speech energy detection is automatically stopped when the first IRE_ENERGY event occurs or activity such as play requests are queued. In other words, it must be explicitly re-started each time that speech energy detection is desired.

irStartSpeechED should not be used when full call classification analysis (CCA) is being used since full CCA includes an option for speech energy detection as a means of detecting an answer. See *irCCA(3IRAPI)* for more information about full CCA.

irStartSpeechED can not be used when speech is being played via the T/R card [as opposed to using an signal processing (SP) card] since speech energy detection does not work while the T/R card is playing speech. If *irStartSpeechED* is used while an SP card is playing speech on the channel, the detection algorithm may be triggered by the played speech being echoed back by the network (an IRE_ENERGY event might be the result of played speech rather than words spoken by a person at the other end of the connection).

Event

For *irStartSpeechED*, IRE_ENERGY is generated when speech energy is detected on the *cid*. *irStopSpeechED* and *irCheckSpeechED* generate no events.

NOTE: An IRE_ENERGY event may also indicate that cessation of ringing has been detected on a T/R card while speech energy detection is turned off. In a future release we may generate a different event for cessation of ringing.

Return Value

For *irStartSpeechED* and *irStopSpeechED*:

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

For *irCheckSpeechED*, IRR_ON and IRR_OFF are returned if speech energy detection is on or off, respectively.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information); in particular, if *irSysError* is ENOSPC, there was other activity still queued for this channel

IRER_UNSUPPORTED if speech energy detection is not supported for the *cid*s telephony type (only T/R channels support speech energy detection)

See Also

irCCA(3IRAPI), IrDEFINES(4IRAPI), IrEVENTS(4IRAPI)

irStop

Name

irStop — Stop recording, playing, saying, dialing, or calling

Synopsis

#include <**irapi.h**>

int irStop (channel_id *cid***);**

Description

The *irStop* function immediately stops recording or playback on a channel indicated by a channel identifier (*cid*). The *irStop* function also aborts a call in progress [see *irCall(3IRAPI)*].

The number of milliseconds played or recorded may be obtained by calling *irGetVCount(3IRAPI)* after IRE_PLAY_DONE or IRE_RECORD_DONE have occurred.

If the channel is in the IRS_PLAY_PENDING, IRS_RECORD_PENDING, IRS_CALL_PENDING, IRS_SAY_PENDING, IRS_PLAY_QUEUED or IRS_SAY_QUEUED library states, it is returned immediately to the IRS_IDLE state and all play, say, call or record requests are removed.

If the *cid* is in the IRS_DIALING, IRS_CALLING, IRS_SAYING, IRS_PLAYING, or IRS_RECORDING library states, an asynchronous *stop* request is sent immediately to the agent for the asynchronous activity. The *cid* library state remains unchanged until the activity specific IRE_<activity>_DONE event occurs.

Note that if play requests are stopped via *irStop* when the channel is in the IRS_PLAY_PENDING state, they cannot be resumed at a later time with *irPlayResume(3IRAPI)*. The play request must be re-sent with the appropriate *irPlay(3IRAPI)* and *irEnd(3IRAPI)* sequences. This holds true for IRS_RECORD_PENDING and *irRecordResume(3IRAPI)*.

Calling *irStop* when the channel is in the IRS_IDLE state has no effect.

Event

No event results directly from the call to *irStop*. Any outstanding activity, such as play, say, record, etc., completes with a IRE_<activity>_DONE event and a IREM_STOPPED level 2 modifier. The *tag* of the event is the tag specified with the function which initiated the activity.

Return Value

IRR_OK is returned if *irStop* has successfully made the stop request.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_BADSTATE if the channel is in an unstoppable state

IRER_DRIVER_ERROR if a driver call failure occurs (check *irSysError* for additional information)

IRER_SYSERROR if an system or driver call failure occurs (check *irSysError* for additional information)

See Also

irPlay(3IRAPI), *irRecord(3IRAPI)*, *irSay(3IRAPI)*, *irDial(3IRAPI)*, *irCall(3IRAPI)*, *IrEVENTS(4IRAPI)*, *IrSTATES(4IRAPI)*.

irTSAlloc

Name

irTSAlloc, irTSFree — Allocate/free a time slot to/from a channel

Synopsis

#include <**irapi.h**>

int irTSAlloc (channel_id *cid***)**

int irTSFree (channel_id *cid***, int** *time_slot***);**

Description

irTSAlloc allocates a single time slot to the channel identifier (*cid*). The allocation of a time slot does not imply anything about its use.

irTSFree frees *time_slot* from ownership by the channel associated with *cid*.

Event

No event results from the call to *irTSAlloc* or *irTSFree*.

Return Value

irTSAlloc returns the time slot allocated if successful. *irTSFree* returns IRR_OK if the time slot is successfully freed.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_RESOURCEBUSY if all time slots are busy

IRER_BADSTATE for *irTSFree* if *time_slot* is currently active for the channel associated with *cid*

IRER_NOTFOUND for *irTSFree* if *time_slot* is not currently owned by the channel associated with the *cid*

See Also

irTSStart(3IRAPI) and *irTSControl(3IRAPI)*.

irTSControl

Name

irTSControl — Control timeslot data on a channel's output

Synopsis

#include <**irapi.h**>

int irTSControl (channel_id *cid***, int** *time_slot***, int** *gain***, int** *flag***);**

Description

irTSControl allows a channel to add or drop an arbitrary timeslot from its voice output. If *flag* is set to IRD_ADD, the channel associated with the *cid* includes the data present on *time_slot* in its output. Output volume on the timeslot input is adjusted by *gain*dB. If *flag* is set to IRD_DROP, the data present on *time_slot* is removed from the channel's output.

A distinct timeslot may only be added once to the channel's output. Calling *irTSControl* with a *cid* for which *time_slot* has already been included on the channel's output might only serve to alter to output volume if the *gain* parameter varies between calls.

Consider using *irHBridge(3IRAPI)* or *irMonitor(3IRAPI)* as alternatives to *irTSControl(3IRAPI)* as these functions meet most typical application needs for timeslot control. *irHBridge(3IRAPI)* has the added benefit of maintaining the Transmission Level Plan (TLP).

Each channel can have at most 7 output timeslots of which one is pre-assigned. So the channel can only add another 6 output timeslots.

Event

No event results from the call to *irTSControl*.

Return Value

irTSControl returns IRR_OK if the the voice data on *time_slot* is successfully added or removed from the channel output.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid or if the *time_slot* is not valid and *flag* is set to IRD_ADD

IRER_NOTFOUND if *time_slot* is not being included in the *cid*s channel output and *flag* is set to IRD_DROP

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_MAXCHAN_TIMESLOTS if the channel has reached the limit for output timeslots (7 maximum per channel).

See Also

irTSAlloc(3IRAPI), *irTSEnd(3IRAPI)*, *irHBridge(3IRAPI)* and *irMonitor(3IRAPI)*.

irTSEnd

Name

irTSEnd, irTSStop — Control play activity on a time slot

Synopsis

#include <**irapi.h**>

int irTSEnd (channel_id *cid***, int** *tag***, int** *time_slot***);**

int irTSStop (channel_id *cid***, int** *time_slot***);**

Description

The *irTSEnd* function terminates the queue of play activities on the specified channel identifier (*cid*) and starts the play on *time_slot*. *tag* is returned with the subsequent event the IRE_TS_DONE event points to [see *IrEVENTS(4IRAPI)*].

Any number of time slot starts may be made per *cid* but only one play may be started per time slot. *time_slot* must be idle and owned by the *cid*. The channel associated with the *cid* for requesting the time slot play may or may not be including the data present on *time_slot* in its output. The activity affects any channel for which *time_slot* is included in its output.

The library state for *cid* is moved from some the IRS_PLAY_QUEUED state to the IRS_IDLE state.

Any play started via *irTSEnd* may be stopped via a call to *irTSStop*.

The use of these functions is discouraged in cases where *irEnd(3IRAPI)* would suffice. *irTSEnd* may only be called from the IRS_PLAY_QUEUED state.

Event

An IRE_TS_DONE event is generated when the activity implied by the queued state runs to completion. Event modifiers are listed in *IrEVENTS(4IRAPI)*.

Return Value

irTSEnd and *irTSStop* return IRR_OK if the request is successfully completed.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is invalid

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

IRER_BADSTATE, for *irTSEnd* only, if the *cid* is not in the IRS_PLAY_QUEUED state

IRER_NORESOURCES, for *irTSEnd* only, if no resources for play exists

IRER_RESOURCEBUSY, for *irTSEnd* only, if all resources for play are busy

IRER_RESTRICTED, for *irTSEnd* only, if resources for play are not in the *cid*'s restricted resource list

IRER_NOTFOUND, for *irTSEnd*, if *time_slot* is not owned by *cid* [see *irTSAlloc(3IRAPI)*] or, for *irTSStop*, if an outstanding activity for *cid* on *time_slot* is not found

Caveat

Preallocation of activity resources via *irReserveRes(3IRAPI)* has no affect on the allocation of activity resources for a time slot start request.

Delayed resource allocation is unimplemented for this service.

See Also

irPlay(3IRAPI), *irTSAlloc(3IRAPI)*and *irTSListen(3IRAPI)*.

irTTTimer

Name

irStartTTTimer, irStopTTTimer — Start/stop the touch-tone input string timer

Synopsis

#include <**irapi.h**>

int irStartTTTimer (channel_id *cid***);**

int irStopTTTimer (channel_id *cid***);**

Description

The *irStartTTTimer* function starts the timer used for touch-tone input string collection based on timer expiration.

The IRE_INPUT_DONE event occurs when certain conditions are satisfied [see *IrEVENTS(4IRAPI)*]. This event may be generated if the touch-tone input timer expires. The time interval starts when *irStartTTTimer* is called. The IRAPI library parameter IRP_TT_PRETIME specifies the time period to wait before the initial touch tone is received while IRP_TT_INTERTIME specifies the time period to wait between two successive touch tones [see *IrPARAMETERS(4IRAPI)*].

If there is no input on the input queue *irStartTTTimer* starts a timer for IRP_TT_PRETIME milliseconds. If there is input on the input queue *irStartTTTimer* starts a timer for IRP_TT_INTERTIME milliseconds. If a touch tone is received before the timeout occurs, the touch-tone timer is canceled. If a IRE_INPUT_DONE event is not generated upon the receipt of a touch tone and *irStartTTTimer* was executed before the last IRE_INPUT_DONE event occurred, a new timer starts with a timeout of IRP_TT_INTERTIME. In general, the IRAPI automatically manages clock timeouts from the point when *irStartTTTimer* is called until IRE_INPUT_DONE is generated, including both predigit and interdigit timeouts.

If a touch-tone timeout occurs before a touch tone arrives and there is no input on the input queue, IRE_INPUT_DONE is generated with modifiers IREM_TT and IREM_INPUT_PRE.

If a touch-tone timeout occurs before a touch tone arrives and if there is input on the input queue, IRE_INPUT_DONE is generated with modifiers IREM_TT and IREM_INPUT_INTER.

The existing values of the IRP_TT_PRETIME and IRP_TT_INTERTIME parameters can be retrieved using *irGetParam(3IRAPI)* while a new value can be assigned to them using *irSetParam(3IRAPI)*. Defaults for each parameter are 5000 milliseconds. These parameters must be set to the desired value before invoking *irStartTTTimer*.

Refer to *irGetInput(3IRAPI)* for other conditions under which IRE_INPUT_DONE event is generated.

Calling *irStartTTTimer* with an outstanding touch-tone timer active cancels the outstanding timer and starts a new timer as defined above.

irStopTTTimer stops the touch-tone timer.

Events

IRE_INPUT_DONE is generated after *irStartTTTimer* is called if the conditions explained above are fulfilled.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an occurs:

IRER_INVALID if the *cid* is invalid

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

See Also

irParam(3IRAPI), *irGetInput(3IRAPI)*, *IrPARAMETERS(4IRAPI)*and *IrEVENTS(4IRAPI)*.

irTalkFiles

Name

irTF2File, irFile2TF — Convert talkfile and phrase number into file name

Synopsis

#include <**irapi.h**>

char *irTF2File (long *talkfile***, long** *phrase_id***);**

int irFile2TF (const char *****filename*, long ***talkfile***, long ***phrase_id***);**

Description

irTF2File converts a *talkfile* and *phrase_id* pair into a UNIX file name. This function is provided to support applications speaking phrases from old talkfile/phrase_id type speech databases. A pointer to the resulting filename is returned. The filename is constructed as follows:

{IRP_SPEECHDIR}/{*talkfile*}/{*phrase_id*}

IRP_SPEECHDIR is a parameter defined in *IrPARAMETERS(4IRAPI)*. *talkfile* and *phrase_id* are the command line arguments to *irTF2File* converted into ascii strings suitable as directory and filenames.

This function may be used as file name input to the *irRecord(3IRAPI)*, *irPlay(3IRAPI)* and *irOpen(3IRAPI)* functions.

irFile2TF converts *filename* into a talkfile and phrase number pair. The "basename" of *filename* is converted into a long placed into the address specified by *phrase_id*. The "basename" of the "dirname" of *filename* is converted into a long placed into the address specified by *talkfile*.

Event

No event results from the call to either *irTF2File* or *irFile2TF*.

Return Value

For *irTF2File*, a pointer to the constructed file name is returned if successful and IRR_NULL is returned if an error occurs.

For *irFile2TF* a IRR_OK is returned if successful and IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid value for *talkfile* or *phrase_id* is passed for *irTF2File*. For *irFile2TF*, if the basename of *filename* and the basename of the dirname of *filename* contain any non-digit characters.

See Also

irPlay(3IRAPI), *irRecord(3IRAPI)* and *irOpen(3IRAPI)*.

irTeleType

Name

irChan2TeleType,irTeleType2TeleClass — Get telephony type and class

Synopsis

#include <**irapi.h**>

int irChan2TeleType (int *channel_nbr***);**

int irTeleType2TeleClass (int *tele_type***);**

Description

The *irChan2TeleType* function returns the telephony type associated with the channel *channel_nbr*. The current possible telephony types (described in *IrDEFINES(4IRAPI)* are IRD_TR, IRD_T1, IRD_PRI, IRD_LST1_DEF, IRD_LST1_GAL, IRD_LST1_ASAI, IRD_ASAI, IRD_VIRT_CHAN. Additional telephony types may be supported by add-on packages.

The *irTeleType2TeleClass* function returns the telephony class associated with the telephony type *tele_type*. The possible telephony classes (described in *IrDEFINES(4IRAPI)* are IRD_ANALOG, IRD_DIGITAL, IRD_VIRTUAL_CLASS, and IRD_UNKOWN_CLASS.

Event

No event results in a call to either of these functions

Return Value

irChan2TeleType returns a telephony type if successful. IRR_FAIL is returned if an error occurs.

irTeleType2TeleClass returns a telephony class if successful or IRD_UNKOWN_CLASS if a determination cannot be made

Error

irError is set as follows if an error occurs:

IRER_INVALID if *channel_nbr* is invalid

IRER_SYSERROR if the Telephony Type cannot be obtained from the Resource Manager (see *irSysError* for additional information)

irTime2Byte

Name

irTime2Byte — Convert an algorithm from milliseconds to bytes

Synopsis

#include <**irapi.h**>

long irTime2Byte (int *algorithm***, long** *time***);**

Description

The *irTime2Byte* function returns the time-equivalence in bytes for a specified *time* interval for a given *algorithm*. *Time* is specified in milliseconds. This function may be used to figure out how much space is required to store a phrase.

The accepted values of *algorithm* are defined in *IrALGORITHMS(4IRAPI)*.

This function is used by an application to convert time to a position in a voice object or to figure out how much space is required to store a phrase.

Event

No event results from the call to *irTime2Byte*.

Return Value

A byte count is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if *algorithm* or *time* are not valid.

Example

See the example in *irByte2Time(3IRAPI)*.

See Also

irByte2Time(3IRAPI), IrALGORITHMS(4IRAPI)

irTimer

Name

irStartTimer, irCancelTimer, irStartPTimer, irCancelPTimer — Start/stop the clock timer

Synopsis

#include <**irapi.h**>

int irStartTimer (channel_id *cid***, int** *interval***, int** *repeat***, int** *tag***);**

int irCancelTimer (channel_id *cid***, int** *tag***);**

int irStartPTimer (int *interval***, int** *repeat***, int** *tag***);**

int irCancelPTimer (int *tag***);**

Description

The *irStartTimer* function starts a timer to go off after *interval* milliseconds (subject to inherent clock granularity - 10 milliseconds for a 486 central processing unit). Multiple clock timers may be started per channel if all timers per channel have unique *tag*s. The *tag* is returned with the subsequent IRE_CLOCK event.

If *repeat* is non-zero, the clock goes off again after *interval* more milliseconds, indefinitely.

irCancelTimer cancels the timer on *cid* with the indicated tag.

irStartPTimer and *irCancelPTimer* are similar to *irStartTimer* and *irCancelTimer* except that the timers are process based rather than channel identifer (cid) based.

Events

IRE_CLOCK occurs when *interval* elapses. If *repeat* is non zero, multiple IRE_CLOCK events occur.

When using *irStartTimer*, the subsequent IRE_CLOCK event's cid element is valid. When using *irStartPTimer*, the subsequent IRE_CLOCK event's cid element is set to IRD_NULL.

Return Value

IRR_OK is returned if *irStartTimer* and *irStartPTimer* are successful.

The number of milliseconds remaining on the timer is returned if *irCancelTimer* and *irCancelPTimer* are successful.

IRR_FAIL is returned for all functions if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the:

- *cid* is invalid
- *interval* is invalid
- *cid* already has a timer with this *tag* (*irStartTimer* only)
- *cid* has no timer with this tag (*irCancelTimer* only)
- calling process already has a timer with this *tag* (*irStartPTimer* only)
- calling process has no timer with this *tag* (*irCancelPTimer* only)

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

See Also

irTTTimer(3IRAPI), *irRecogTimer(3IRAPI)* and *IrEVENTS(4IRAPI)*.

irTrace

Name

irTrace, irQTrace, irTraceP, irQTraceP, irTRACEPROC_CHK, irTRACECHAN_CHK, irTrace_Put, irSetTraceArea, irSetTraceQkey, irSetTraceChan, irSetTraceLevel, irSetTraceLogMode, irSetTraceDateMode — Append data to the trace log

Synopsis

#include <**irapi.h**>

void irTrace (int *channel***, unsigned long** *level***, unsigned long** *area***, const char *** *fmt***, /*** *varargs****/);**

void irQTrace (int *channel***, unsigned long** *level***, unsigned long** *area***, const char *** *str***);**

void irTraceP (int *channel***, unsigned long** *level***, unsigned long** *area***, const char *** *fmt***, /*** *varargs****/);**

void irQTraceP (int *channel***, unsigned long** *level***, unsigned long** *area***, const char *** *str***);**

int irTRACECHAN_CHK (int *channel***, unsigned long** *level***, unsigned long** *area***);**

int irTRACEPROC_CHK (int *channel***, unsigned long** *level***, unsigned long** *area***);**

void irTrace_Put (const char * *str***);**

void irSetTraceChan (int *channel***, int** *flag***);**

void irSetTraceQkey (ir_key_t *qkey***, int** *flag***);**

void irSetTraceArea (unsigned long *area***, int** *flag***);**

void irSetTraceLevel (unsigned long *level***, int** *flag***);**

void irSetTraceLogMode (int *flag***);**

void irSetTraceDateMode (int *flag***);**

Description

irTrace formats and logs data into the trace shared memory buffer or to the trace log if tracing is on for the specified *channel* at the indicated *level* for *area*. Any level of tracing is allowed.

irTraceP formats and logs data into the trace shared memory buffer or to the trace log if tracing is on for the indicated *level* for *area* and for the calling process, or if the *channel* is valid and tracing is on for that channel. Any level of tracing is allowed. If *channel* is -1, the trace message does not display the channel number.

irQTrace and *irQTraceP* are "quick" versions of *irTrace* and *irTraceP*, respectively. *irQTrace* and *irQTraceP* are macros that perform all the necessary checks, and then only call a trace subroutine if necessary. However, *irQTrace* and *irQTraceP* do not accept variable arguments.

The *irTRACECHAN_CHK* macro returns non-zero if channel-level tracing is on for the given *channel*, *level*, and *area*.

The *irTRACEPROC_CHK* macro returns non-zero if process-level tracing is turned on for this process' given *channel*, *level*, and *area*.

irTrace_Put writes the given *str* into the trace shared memory buffer or to the trace log unconditionally. It is recommended that this function be used in conjunction with the *irTRACECHAN_CHK* and *irTRACEPROC_CHK* macros.

By default, trace messages are logged to the trace shared memory buffer. To direct the trace messages to the trace log file, call *irSetTraceLogMode* with a argument of **IRD_ON**. To direct the trace messages to the trace shared memory buffer, call *irSetTraceLogMode* with a argument of **IRD_OFF**.

By default, trace messages are not timestamped when they are placed in the trace shared memory buffer. To turn on the timestamping of every trace message when it is placed in the trace buffer, call *irSetTraceDateMode* with a argument of **IRD_ON**. To turn off the timestamping of every trace message, call *irSetTraceDateMode* with a argument of **IRD_OFF**.

fmt is a *printf(3s)* style formatting string and *varargs* are the arguments required to format the string. All trace messages logged with *irTrace* are logged with the TRACE001 message. By default, TRACE001 is assigned to the *trace* logger destination. The TRACE001 log message has the following format:

`"TRACE001 -- CH %3d<<chan,D>> (TRACE_GEN) %s<<message,S>>"`

Here, `%3d<<chan>>` is the *channel* argument and `%s<<message,S>>` is *fmt* formatted with *varargs*.

Trace messages are logged only when some process either via *trace(1VIS)* or *irSetTrace<Type>* send tracing on messages matching the arguments set to *irTrace.*

A process must be registered with *irRegister(3IRAPI)* before tracing for that process may begin.

irSetTraceChan activates or deactivates trace messages for *chan* as indicated by *flag*. *flag* may be either **IRD_ON** or **IRD_OFF**.

irSetTraceArea activates or deactivates trace messages for *area* as indicated by *flag*. *flag* may be either **IRD_ON** or **IRD_OFF**.

irSetTraceLevel activates or deactivates trace messages for *level* as indicated by *flag*. *flag* may be either **IRD_ON** or **IRD_OFF**.

irSetTraceQkey activates or deactivates trace messages for *qkey* as indicated by *flag*. *flag* may be either **IRD_ON** or **IRD_OFF**.

areas and *levels* are bit maps used to allow *trace(1VIS)* to be more selective in terms of trace output. This means the higher the level, the greater the level of detail on trace output. Level 32 messages include the greatest level of detail, and level 1, the least. Trace levels are defined by the IRD_TRACE_LEVEL_1 through IRD_TRACE_LEVEL_16 for user defined levels, these levels occupy bits 0 through 15 in the *level* bit mask. IRAPI applications may use these levels and attach whatever meaning to these levels that suits them. *trace(1VIS)* supports tracing on these user defined levels. Trace levels 17 through 32 are reserved for internal IRAPI library functions and VIS processes. It is not recommended that user applications generate trace messages in the reserved levels. However, applications may use *irSetTraceLevel* to enable tracing on these levels.

Trace areas are defined by the IRD_TRACE_AREA_1 through IRD_TRACE_AREA_16 for user defined areas, these areas occupy bits 0 through 15 in the *area* bit mask. IRAPI applications may use these areas and attach whatever meaning suits them to these areas. *trace(1VIS)* supports tracing on these user defined areas. Trace areas 17 through 32 are reserved for internal IRAPI library functions and VIS processes. Defines symbols for these areas are found in *IrDEFINES(4IRAPI)* as IRD_TRACE_AREA_<area>, where <area> is a two letter mnemonic for the particular area. It is not recommended that user applications generate trace messages in the reserved areas. However, applications may use *irSetTraceArea* to enable tracing on these areas.

Tracing may be done on these areas and levels through the *trace(1VIS)* command.

Event

No event results from the call to *irTrace*.

Return Value

none

Error

none

See Also

trace(1VIS), *IrDEFINES(4IRAPI)*.

irUngetInput

Name

irUngetInput — Place a character on the input queue

Synopsis

#include <**irapi.h**>

int irUngetInput (channel_id *cid***, char ***i_buf***);**

Description

The *irUngetInput* function places the character string *i_buf* onto the input queue of the channel specified by channel identifier (*cid*). Input characters are placed in queue in last-in-first-out (LIFO) order – they are the first out [see *irGetInput(3IRAPI)*].

Event

No event results from the call to *irUngetInput*.

Return Value

The number of characters placed on the input queue is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *cid* is not valid

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

See Also

irGetInput(3IRAPI), irFlushInput(3IRAPI)

irVfd2Fd

Name

irVfd2Fd — Convert a voice file descriptor to a file descriptor

Synopsis

#include <**irapi.h**>

int irVfd2Fd (vf_descriptor *vfd***);**

Description

The *irVfd2Fd* function converts a voice file descriptor (vfd), obtained by a previous *irOpen(3IRAPI)*, to a UNIX file descriptor.

Vfd is obtained by *irOpen(3IRAPI)*, and discarded by *irClose(3IRAPI)*.

The file descriptor returned by *irVfd2Fd* is opened with the same value of *oflag* used in the call to *irOpen(3IRAPI)*. However, the O_CREAT, O_TRUNC and O_EXCL flags is ignored. The file descriptor points to the position in the file previously set with *irLSeek(3IRAPI)* however, the offset is in bytes rather than milliseconds of speech.

Note that voice file descriptors are not updated with respect to position as the files they represent are played or recorded. The position of a voice file descriptor is modified only through the use of the *irLSeek(3IRAPI)* function. This is because voice file descriptors may be shared across many channels within the context of a single process.

This function allows an application access to UNIX system calls, such as *fstat(2)*.

Event

No event results from the call to *irVfd2Fd*.

Return Value

A UNIX file descriptor (*fd*) is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if the *vfd* is invalid

IRER_UNSUPPORTED if the file referred to by *vfd* is coded in an unknown algorithm type. Note: This error only occurs if you have previously used *irLSeek(3IRAPI)* on the vfd. Converting time to bytes on the voice file pointer is not possible if the coding algorithm cannot be determined.

IRER_SYSERROR if a system or driver call failure occurs (check *irSysError* for additional information)

irWCheck

Name

irWCheck — Wait and get event information

Synopsis

#include <**irapi.h**>

int irWCheck (ir_event_t * *pir_event***);**

Description

The *irWCheck* function combines *irWait(3IRAPI)* and *irCheck(3IRAPI)* as a convenience to the user. *irWCheck* is a blocking function used to access events.

The events and event structure (*ir_event_t*) are specified in *IrEVENTS(4IRAPI)*.

irWCheck returns information about only one event. If multiple events are queued, *irWCheck* returns the first one.

irWCheck returns if there is an enabled event on the event queue. It blocks if there is no enabled event on the event queue.

irWCheck never returns for IRE_NULL events.

Event

No event results from the call to *irWCheck*.

Return Value

irWCheck returns the *event_id* if successful [see *IrEVENTS(4IRAPI)*].

IRR_FAIL is returned if an error occurs.

Error

Error strategies as defined for *irWait* apply for *irWCheck*.

Caveat

The application must call function *irWCheck* or *irWait(3IRAPI)* soon after initiating one or more asynchronous speech functions to ensure proper handling of the speech at the driver level.

See Also

irWait(3IRAPI), irCheck(3IRAPI)

irWait

Name

irWait — Wait for a voice event

Synopsis

#include <**irapi.h**>

int irWait ( );

Description

The *irWait* function waits for an event to occur, then returns. *irWait* is the event notification mechanism of the IRAPI. Significant amounts of time, such as those introduce through system calls, should not elapse between calls to *irWait*.

The *irCheck(3IRAPI)* function is typically called after *irWait* to obtain a description of the event.

Event

No event results from the call to *irWait*.

Return Value

IRR_OK is returned if an event occurred.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_SYSERROR if a system call failure occurs (check *irSysError* for additional information)

IRER_NOREGISTER if the process has not been registered via a call to *irRegister(3IRAPI)*

Caveat

The application must *irWait* soon after initiating one or more asynchronous speech functions to ensure proper handling of the system events at the driver level.

See Also

irCheck(3IRAPI), irSetEvent(3IRAPI), irWCheck(3IRAPI).

iraAddAD

Name

iraAddADChannel — Add an entry in the Application Dispatch channel table

iraAddADDnisani — Add an entry in the Application Dispatch DNIS/ANI table

Synopsis

#include <**irapi-ad.h**>

int iraAddADChannel (int *channel***, int** *disp_mode***, const char *** *reg_file***);**

int iraAddADDnisani (const IRA_STR_RANGE *****dnisrange*, const IRA_STR_RANGE *****anirange***, const char *****reg_file***);**

Description

The *iraAddADChannel* and *iraAddADDnisani* functions allow users to add entries to the AD channel table and the AD DNIS/ANI table respectively [see *IRAPI-AD(4IRAPI-AD)*].

The *iraAddADChannel* function adds the application described by the registration file *reg_file* for channel *channel* and dispatch mode *disp_mode* into the Application Dispatch channel table.

See *IRAPI-AD(4IRAPI)* for a description of the *AD_APPL* structure. The *disp_mode* argument specifies the dispatch mode for the application. Possible values are *IRD_AD_STANDARD* or *IRD_AD_STARTUP* [see *IRAPI-AD(4)*].

The *iraAddADDnisani* function adds the application described by the registration file *reg_file* for the dnis range *dnisrange* and ani range *anirange* into the Application Dispatch DNIS/ANI table.

The *start* value of a range must be less than or equal to the *end* value of the range. See *IrAD-API(4IRAPI)* for a complete description of the *IRA_STR_RANGE* structure.

The global parameter IRP_AD_MODE must be set to IRD_AD_READWRITE or both functions will return an error.

The global parameters IRP_AD_CHANNEL_TABLE and IRP_AD_DNISANI_TABLE contain the full pathnames of the AD Channel and AD DNIS/ANI tables, respectively.

Return Value

IRR_OK is returned if the request is sucessful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid argument is passed in

IRER_OVERFLOW if the Application Dispatch DNIS/ANI table is full

IRER_BADRANGE if an invalid range was passed into the function *iraAddADDnisani*

IRER_PERMISSION if the global parameter IRP_ADAPI_MODE is not set to IRD_ADAPI_READWRITE

IRER_SYSERROR if a system error occurred (*irSysError* is set to the corresponding *errno*)

See Also

iraRemoveAD(3IRAPI-AD), IRAPI-AD(4IRAPI-AD), IrDEFINES(4IRAPI)

iraInitAD

Name

iraInitADTables — Initialize both the AD Channel and DNIS/ANI tables

iraInitADChannel — Initialize the AD Channel table

iraInitADDnisani — Initialize the AD DNIS/ANI table

Synopsis

#include <**irapi-ad.h**>

int iraInitADTables (int *numchans***, int** *numdnisani***);**

int iraInitADChannel (int *numchans***);**

int iraInitADDnisani (int *numdnisani***);**

Description

Functions *iraInitADTables*, *iraInitADChannel* and *iraInitADDnisani* allow users to initialize the both the AD Channel and DNIS/ANI tables, just the AD Channel table, or just the AD DNIS/ANI table, respectively (see *IRAPI-AD(4IRAPI-AD)*).

Function *iraInitADTables* creates and initializes an AD Channel table with *numchans* entries. It also creates and initializes an AD DNIS/ANI table with *numdnisani* entries.

Function *iraInitADChannel* creates and initializes an AD Channel table with *numchans* entries.

Function *iraInitADDnisani* creates and initializes an AD DNIS/ANI table with *numdnisani* entries.

The global parameter IRP_AD_MODE must be set to IRD_AD_READWRITE or both functions will return an error.

The global parameters IRP_AD_CHANNEL_TABLE and IRP_AD_DNISANI_TABLE contain the full pathnames of the AD Channel and AD DNIS/ANI tables, respectively.

Return Value

IRR_OK is returned if the request is sucessful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid argument is passed in

IRER_PERMISSION if the global parameter IRP_ADAPI_MODE is not set to IRD_ADAPI_READWRITE

IRER_SYSERROR if a system error occurred, *irSysError* is set to the corresponding *errno*

See Also

IRAPI-AD(4IRAPI-AD), irAPI.rc(4IRAPI)

iraQueryAD

Name

iraQueryADTables — Query Application Dispatch tables

iraQueryADDnisani — Query Application Dispatch DNIS/ANI table

Synopsis

#include <**irapi-ad.h**>

int iraQueryADTables(channel_id *cid***, int** *mode***, AD_APPL *****appl***);**

int iraQueryADDnisani(int *channel***, int** *mode***, const char *****dnisstring***, const char *** *anistring***, AD_APPL *****appl***);**

Description

The *iraQueryADTables* function allows an application to determine which application should be dispatched on the *cid* for dispatch mode *mode*. *appl* must be the address of a AD_APPL structure. The *iraQueryADTables* function first looks at the entry in the channel-based AD table for the channel belonging to the *cid* and dispatch mode *mode*. If the entry is not NULL and the service name is not "*DNIS_SVC", then the entry is copied into the AD_APPL structure pointed to by *appl*. If the entry is NULL or the service name is "*DNISANI_SVC", *iraQueryADTables* will search sequentially through the DNIS/ANI-based AD table for a entry matching the the DNIS and ANI of channel belonging to the *cid*. If an entry that matches both the DNIS and ANI is found, the matching entry is copied into the AD_APPL structure pointed to by *appl*. The DNIS and ANI of channel belonging to the *cid* is obtained by using *irGetIEs(3IRAPI)* (see *irIE(3IRAPI)*).

The *iraQueryADDnisani* function performs the same function as *irAQueryADTables*, but instead of using *irGetIEs(3IRAPI)* to obtain the dnis and ani strings, the dnis and ani string are passed into the function as arguments.

Return Value

IRR_OK is returned if the functions are successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid argument is passed in

IRER_NOTFOUND if no entry is found that matches the *dnisstring* and *anistring*

IRER_SYSERROR if a system error occurred, *errno* is set accordingly

See Also

IRAPI-AD(4IRAPI-AD), irIE(3IRAPI)

iraReadAD

Name

iraOpenADTables, iraReadADChannel, iraReadADDnisani, iraCloseADTables — Open, read, and close the Application Dispatch table

Synopsis

#include <**irapi-ad.h**>

int iraReadADChannel(int *chan***, int** *mode* **, AD_APPL *****p_appl***)**

int iraReadADDnisani(AD_DNISANI_ENTRY *****p_ad_dnisani_entry*)**

int iraRewindADDnisani()

Description

This group of functions allow applications to read the AD tables one entry at a time.

The *iraReadADChannel* function copies the application for channel *chan* and dispatch mode *mode* from the channel-based AD table into AD_APPL variable at address *p_appl*. The application developer must allocate storage for the AD_APPL variable.

The *iraReadADDnisani* function reads the DNIS/ANI AD table sequentially, starting at the first entry. The first call to *iraReadADDnisani* returns the first entry of the DNIS/ANI-based table and each subsequent call to *iraReadADDnisani* returns each subsequent DNIS/ANI-based entry. *iraReadADDnisani* copies the entry from the DNIS/ANI-based AD table into AD_DNISANI_ENTRY variable at address *p_ad_dnisani_entry*. The application developer must allocate storage for the AD_DNISANI_ENTRY variable.

The *iraRewindADDnisani* function can be used force the next call to *iraReadADDnisani* to read the first entry of the DNIS/ANI-based table.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid argument is passed in

IRER_ENDOFTABLE when the end of AD DNIS/ANI table is reached

IRER_SYSERROR if a system error occurred, *errno* is set accordingly.

See Also

iraAddAD(3IRAPI-AD), iraRemoveAD(3IRAPI-AD), iraQueryAD(3IRAPI-AD), IRAPI-AD(4IRAPI-AD)

iraRemoveAD

Name

iraRemoveADChannel — Remove an entry from the Application Dispatch channel table

iraRemoveADDnisani — Remove an entry from the Application Dispatch DNIS/ANI table

Synopsis

#include <**irapi-ad.h**>

int iraRemoveADChannel(int *chan***, int** *mode***)**

int iraRemoveADDnisani(const IRA_STR_RANGE *****dnisrange*, const IRA_STR_RANGE *****anirange***)**

Description

The *iraRemoveADChannel* function removes the entry specified by *chan* for dispatch mode *mode* from the Application Dispatch channel table.

The *iraRemoveADDnisani* function removes the entry specified by the dnis range *dnisrange* and ani range *anirange* into the Application Dispatch DNIS/ANI table.

See *IRAPI-AD(4IRAPI)* for a description of the *IRA_STR_RANGE* structure.

The global parameter **IRP_AD_MODE** must be set to **IRD_AD_READWRITE** or both functions return an error.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid argument is passed in

IRER_NOTFOUND if the specified entry was not found

IRER_PERMISSION if the global parameter IRP_AD_MODE is not set to IRD_AD_READWRITE

IRER_SYSERROR if a system error occurred, *errno* is set accordingly

See Also

iraAddAD(3IRAPI-AD), iraReadAD(3IRAPI-AD), iraQueryAD(3IRAPI-AD), IRAPI-AD(4IRAPI-AD)

iraReadReg

Name

iraReadRegFile, iraReadRegFP — Read the contents of an IRAPI Application Registration file into an AD_APPL structure

Synopsis

#include <**irapi-ad.h**>

int iraReadRegFile(const char *_reg_file_, AD_APPL *_p_appl_)

int iraReadRegFP(FILE *_fp_, AD_APPL *_p_appl_)

Description

These two functions read the contents of a registration file and place the information into a AD_APPL structure pointed to by *p_appl*.

The *iraReadRegFile* function takes a filename of a registration file and uses the function *iraRegFilePath(3IRAPI)* to translate the filename into a full pathname to the registration file. It then opens the file and calls *iraReadRegFP* to read the file.

The *iraReadRegFP* function takes a FILE pointer to an already open registration file.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid argument is passed in

IRER_SYSERROR if a system error occurred, *irSysError* is set to *errno* accordingly

See Also

iraRegFilePath(3IRAPI-AD), iraWriteReg(3IRAPI), IRAPI-AD(4IRAPI-AD)

iraRegFilePath

Name

iraRegFilePath — Create the full pathname of an IRAPI Application Registration file

Synopsis

#include <**irapi-ad.h**>

int iraRegFilePath(char * *dest*, **char *** *file*)

Description

Given the filename *file*, this function will place the full pathname of the IRAPI Application Registration file for *file* in the string pointed to by dest.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if the string pointed to by *file* is NULL or too long.

See Also

iraReadReg(3IRAPI-AD), iraWriteReg(3IRAPI), IRAPI-AD(4IRAPI-AD)

iraSetStrRange

Name

iraSetStrRange — Set the start and end values of a string range

Synopsis

#include <**irapi-ad.h**>

int iraSetStrRange(IRA_STR_RANGE * *strrange*, **char *** *start*,
char * *end*)

Description

This function takes a *start* and an *end* digit strings and sets them in an IRA_STR_RANGE structure.

The IRA_STR_RANGE structure is used to assign DNIS and ANI based applications to the Application Dispatch tables.

The strings "any" and "all" are also accepted as valid strings for *start* and/or *end*.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set to IRER_INVALID if *start* and/or *end* contained characters that are not digits, as defined by the *isdigit(3)* function, and are not the strings "any" or "all".

See Also

ctype(3C), iraWriteADDnisani(3IRAPI-AD), IRAPI-AD(4IRAPI-AD)

iraWriteAD

Name

iraWriteRegFile, iraWriteRegFP — Write the contents of an AD_APPL structure into an IRAPI Application Registration file

Synopsis

#include <**irapi-ad.h**>

int iraWriteRegFile(AD_APPL * *p_appl***)**

int iraWriteRegFP(FILE * *fp***, AD_APPL *** *p_appl***)**

Description

These two functions write the contents of an AD_APPL structure, pointed to by *p_appl* into the corresponding registration file.

The *iraWriteRegFile* function takes a pointer to an AD_APPL structure and then uses the function *iraRegFilePath(3IRAPI)* to translate the *reg_file* field into a full pathname of the registration file. It then opens the file and calls *iraWriteRegFP* to write the file.

The *iraWriteRegFP* function takes a FILE pointer to an already open registration file and writes the contents of the AD_APPL structure pointed to by *p_appl* to the file.

Return Value

IRR_OK is returned if the request is successful.

IRR_FAIL is returned if an error occurs.

Error

irError is set as follows if an error occurs:

IRER_INVALID if an invalid argument is passed in

IRER_SYSERROR if a system error occurred, *irSysError* is set to *errno* accordingly

See Also

iraRegFilePath(3IRAPI-AD), iraReadReg(3IRAPI-AD), IRAPI-AD(4IRAPI-AD)

IRAPI-AD

Name

IRAPI-AD — IntuityResponse API for Application Dispatch

Synopsis

#include <**irapi-ad.h**>

Terminology

default owner — The default owner of a channel is the process which will become the owner of a channel when the current owner relinquishes ownership of the channel through the *irDeinit(3IRAPI)* library function.

application — An IRAPI application consists of a service name, a process name, and a process type. All information about an application is stored in the application registration file under /vs/trans. Users refer to IRAPI applications by using the name of the registration file.

dispatch mode — Within the AD Channel table, there are two different dispatch modes to which an application can be assigned, Standard and Startup, which are described below:

Standard — The Standard application for a channel is dispatched whenever the AD process receives an IRE_EXEC event for that particular channel, or whenever the AD process receives an IRE_NEWCALL event for that particular channel and there is no Startup application for that channel.

Startup — The Startup application for a channel is dispatched whenever the AD process receives an an IRE_NEWCALL event for that particular channel. The assignment of a Startup application for a channel is optional and only needed for certain situations. Namely, (add examples)

service name — The name of the service, traditionally the name of the TSM script.

process name	For transient processes, this is the full pathname of the process. For permanent processes, this is the name the process passes to irRegister(3IRAPI).
process type	There are two different types of IRAPI processes, Permanent, and Transient, which are described below:
Permanent	This is a IRAPI process that doesn't exit. Examples of Permanent IRAPI applications include TSM, AD, and DIPs.
Transient	This is a IRAPI process that exits when it has completed its task.

Description

The IntuityResponse API for Application Dispatch (IRAPI-AD) is a set of library functions to add, remove, query, and read the Application Dispatch tables. There are two AD tables, one for applications that are started based on the channel of the call (channel-based applications) and the other for applications that are started based on the DNIS and ANI of the call (DNIS/ANI-based applications). By using the IRAPI-AD, users can determine which application should be started for a given channel and event or DNIS and ANI combination.

The AD table for the channel-based applications is a two-dimensional array of *AD_APPL* elements, indexed by channel number and dispatch mode. It is used to map channel number and dispatch modes to applications. For every channel, there are two modes for which an application can be dispatched, the Standard mode and the Startup mode. Normally, only the Standard application will be used, but there are some situations where it is desirable to have a different application dispatched upon call startup.

The AD table for the DNIS/ANI-based applications is a linked list of *AD_DNISANI_ENTRY* elements, one for each DNIS/ANI application. The DNIS/ANI entry in the AD table will contain a range for the DNIS and a range for the ANI, each specified in an IRD_STR_RANGE element. The linked list is sorted most specific to most general based on the specified DNIS range. Applications with duplicate DNIS ranges are sorted most specific to most general based on the specified ANI range. The default entry will be handled separately. A call must be included within both the DNIS and the ANI ranges to be considered a match. If the DNIS and ANI of an incoming call do not match any of the DNIS/ANI ranges, the call will be handled by the the default entry. To specify the default entry, users should specify both DNIS and ANI ranges as *any*.

Since the DNIS/ANI table is searched sequentially, if two DNIS/ANI entries match a call, only the first matching entry in the DNIS/ANI table will be used.

Use the functions *iraAddADChannel* and *iraAddADDnisani* to add a AD table entry to the corresponding AD table. The *iraRemoveADChannel* and *iraRemoveADDnisani* functions remove an entry for the corresponding AD table.

The AD tables are stored on disk in the files **/vs/data/ad_channel_table** and **/vs/data/ad_dnisani_table**. The IRAPI-AD library functions will use the system call *mmap(2)* to memory map the files into memory, thereby allowing automatic updates of the disk files and allowing each function to access the AD tables directly.

The IRAPI-AD allows the user to modify the AD tables only if the global parameter **IRP_AD_MODE** is set to **IRD_AD_READWRITE**, otherwise calls to *iraAddADChannel*, *iraAddADDnisani*, *iraRemoveADChannel*, and *iraRemoveADDnisani* will return an error. The integrity of the AD tables will be ensured by using a semaphore to control access to the tables. The IRAPI-AD library will hide all the details, (mmap'ing, semaphores, etc), of accessing the AD tables.

Given a channel and a dispatch mode, the *iraQueryADDispatch* function will return the application that should be dispatched based on the contents of the AD tables and the DNIS and ANI of the call. While this function is used primarily by AD, any application can use it. This allows an application to dispatch another application on a channel itself, instead of requiring AD to do it.

Defines

The following are the defines used by the IRAPI-AD

IRD_AD_STANDARD	Used as a argument to iraAddADChannel and iraReadADChannel to indicate the application to be added or read is the Standard mode application.
IRD_AD_STARTUP	Used as a argument to iraAddADChannel and iraReadADChannel to indicate the application to be added or read is the Startup mode application. **IRD_AD_READONLY** Value of the IRP_AD_MODE parameter.
IRD_AD_READWRITE	Value of the IRP_AD_MODE parameter.
IRD_AD_MAXDISPMODES	Maximum number of dispatch modes for applications.
IRD_AD_MAX_DNIS_ANI	Maximum number of DNIS/ANI based applications supported.

Data Structures

The following structures are used to pass information by the *iraAddADDnisani()* and *iraReadADDnisani()* functions.

```
typedef AD_DNISANI_ENTRY struct ad_dnisani_entry

typedef struct ad_dnisani_entry {
    IRA_STR_RANGE dnis;
    IRA_STR_RANGE ani;
    AD_APPL       appl;
} AD_DNISANI_ENTRY;
```

The *ad_dnisani_entry* fields are:

dnis	An IRA_STR_RANGE element containing the dnis range for this entry.
ani	An IRA_STR_RANGE element containing the ani range for this entry.
appl	An *ad_appl* structure containing the application for this entry.

The following structure is used to pass information by the *iraAddADChannel()*, *iraReadADChannel()*, and *iraQueryADDispatch()* functions.

```
typedef struct ad_appl {
    char    service[IRD_SERVICE_NAME_LEN];
    char    process[IRD_PROCESS_NAME_LEN];
    char    register_file[IRD_MAX_FILE_LEN];
    int     type;
    unsigned long attributes;
    time_t  modtime;
} AD_APPL;
```

The *ad_appl* fields are:

service	The service name of the application.
process	The process name of the process that provides the application.
register_file	The registration filename that describes the application.
type	The IRAPI process type of process that provides the application. This field must be either *IRD_PERMANENT* or *IRD_TRANSIENT*.
attributes	The attributes for this application.
modtime	The modification time of the registration file stored as UNIX time.

IrALGORITHMS

Name

IrALGORITHMS — IRAPI speech recording algorithms

Synopsis

#include <**irapi.h**>

Description

The following speech recording algorithms are supported:

IRA_S16	16 K Sub-Band Coding (SBC)
IRA_S24	24 K SBC
IRA_P	64 K Pulse Code Modulation (PCM)
IRA_A_CS16	16 K Intuity CONVERSANT VIS Adaptive Delta Pulse Code Modulation (ADPCM)
IRA_A_CS32	32 K Intuity CONVERSANT VIS ADPCM

The following algorithm flags are provided to set options when recording voice. These flags should be bitwise OR'd with the algorithm type when setting the IRP_RECORD_ALGO parameter before recording is started.

IRF_NO_SIL	If set, do not trim silence from the recorded speech.
IRF_NO_ACG	If set, do not perform automatic gain control.

See Also

IrPARAMETERS(4IRAPI).

IrDEFINES

Name

IrDEFINES — IRAPI library defines

Synopsis

#include <**irapi.h**>

Description

Following is the list of supported defines that can be used in different IRAPI functions:

IRD_ACTIVE	Indicates an active channel service state
IRD_ADD	Signifies a flag values to start/add
IRD_ALLOW	Indicates that when a channel is requested from this application, the channel should be granted if the service state of the channel is IRD_INACTIVE
IRD_ANALOG	Specifies the telephony class that includes all analog telephony types (for example, IRD_TR, IRD_ASAI)
IRD_ANI	Specifies the information element ANI (calling number for inbound call)
IRD_ASAI	Specifies the telephony type for Tip/Ring (T/R) lines used with ASAI
IRD_BEARER_CAP	Specifies the information element BEARER_CAP (bearer capability for outbound call)
IRD_BLIND_CCA	Indicates outbound calls are to be made without call classification analysis (CCA)
IRD_BLOCKFOREVER	Indicates that when resources are requested, the channel blocks indefinitely for resource availability, if needed
IRD_CHAN_OOS	Indicates a service state of channel out-of-service
IRD_CONDITIONAL	Indicates that when a channel is requested from this application, the ownership of the channel is passed to the requester when the current application *irDeinit(3IRAPI)*s the channel

IRD_DEFAULT	Assumes different default values for selected parameters based on the context. For example, IRP_OUTCALL_ANSDET can be set to IRD_DEFAULT to indicate that the most appropriate form of answer detection should be used based on the telephony type.
IRD_DENY	Indicates that when a channel is requested from this application, the request is denied
IRD_DIALTYPE_DP	Specifies dial pulse dialing
IRD_DIALTYPE_MF	Specifies multi-frequency dialing (not yet supported)
IRD_DIALTYPE_TT	Specifies touch-tone dialing
IRD_DIGITAL	Specifies the telephony class that includes all digital telephony types (for example, IRD_T1, IRD_PRI, IRD_LST1*)
IRD_DNIS	Specifies the information element DNIS (dialed number identification service)
IRD_DROP	Specifies a flag value to stop/drop
IRD_FALSE	Specifies logical false
IRD_FLASH_AND_WAIT	Indicates *irFlash* should wait briefly after generating a flash to allow the switch to get ready for dialing of digits (currently applies only to Line Side T1)
IRD_FLASH_NO_WAIT	Indicates *irFlash* should generate a flash and immediately generate IRE_FLASH_DONE event
IRD_FLEX_WORD	Indicates FlexWord recognition
IRD_FULL_CCA	Indicates that CCA is to be done using the CCA facilities of a dedicated SP circuit card
IRD_IMMEDIATE	Indicates that when resources are requested, unavailability should be indicated immediately
IRD_INACTIVE	Indicates an inactive channel service state
IRD_INBOUND_SERVICE	Specifies the information element INBOUND_SERVICE (for service type on an inbound call)
IRD_INFLECTION_FALLING	Indicates falling inflection
IRD_INFLECTION_NONE	Indicates no inflection
IRD_INFLECTION_RISING	Indicates rising inflection

IRD_INFLECTION_TOTAL
Indicates total inflection

IRD_INVALID Indicates an invalid channel number

IRD_IRAPI Indicates a message is to be routed by a channel

IRD_LST1_ASAI Indicates the telephony type for Line Side T1 (LST1) used with ASAI for DEFINITY switch

IRD_LST1_DEF Indicates the telephony type for LST1 for DEFINITY switch (without ASAI)

IRD_LST1_GAL Indicates the telephony type for LST1 for Galaxy ACD

IRD_MAXCHAN Indicates the upper limit on the number of real channels; use *irNumChans(IRD_REAL)* to determine the actual number of channels

IRD_MAXVCHAN Indicates the upper limit on the number of virtual channels; use *irNumChans(IRD_VIRTUAL)* to determine the actual number of channels

IRD_MAXDATABUFFER Indicates the maximum data buffer which may be passed on *irExec(3IRAPI)*

IRD_MAX_ANI_DIGITS Indicates the maximum length of an IRD_ANI digit string

IRD_MAX_APP_NAME Indicates the maximum length of an IRAPI application name

IRD_MAX_DNIS_DIGITS Indicates the maximum length of an IRD_DNIS digit string

IRD_MAX_FILE_LEN Indicates the maximum voice file length allowed by the *irFPlay(3IRAPI)*, *irFRecord(3IRAPI) , irTF2File(3IRAPI)* and *irOpen(3IRAPI)* functions.

IRD_MAX_REDIRECT_DIGITS
Indicates the maximum length of an IRD_REDIRECTING digit string.

IRD_MAX_RESERVE_RES
Indicates the maximum number of preallocated resources per *cid*.

IRD_MEXICANSPANISH Indicates Mexican Spanish language

IRD_NULL Used as a general purpose null pointer

IRD_OFF Specifies a logical "off"

IRD_ON Specifies a logical "on"

IRD_OUTBOUND_ANI Specifies the Information element OUTBOUND_ANI (calling number of an outbound call)

IRD_PERMANENT	Specifies an IRAPI process type which typically supports multiple channels and continues running after calling *irDeinit(3IRAPI)*
IRD_PRI	Specifies the ISDN Primary Rate Interface (PRI) telephony type (all types) (Add-on packages may change the behavior of this telephony type)
IRD_READONLY	Indicates a read only status of some object
IRD_READWRITE	Indicates a read/write status of some object
IRD_REAL_CHANS	Specifies an abstract channel equipment group; typically used as first argument to *irInitGroup(3IRAPI)* when requesting any real channel
IRD_REAL	Indicates an abstract channel type that includes all channels with network interfaces
IRD_REDIRECTING	Indicates an Information element for redirecting digits
IRD_REMOTE_CLEAR	Indicates a service state where the network has completely cleared a call, but the application has not yet disconnected (specific to PRI)
IRD_REMOTE_DISCON	Indicates a service state where remote switch has disconnected from an active call (specific to PRI)
IRD_RINGING	Indicates a service state of ringing
IRD_SERVICE_TYPE	Specifies the information element SERVICE_TYPE (for an outbound call)
IRD_SIMPLE_CCA	Indicates that CCA for outbound calls is to be done by the channel telephony hardware
IRD_SP_ECHO	Indicates an echo cancellation type (currently the only one)
IRD_SP_VOICE	Indicates a voice type where SP circuit card provides voice operations
IRD_SPEECH_BUF_SIZE	Indicates the size of a speech buffer used for buffer recording
IRD_TALK_VOICE	Indicates a voice type where the T/R circuit card provides voice operations
IRD_TDM_TS	Indicates a voice type where voice operations are performed on a TDM bus timeslot.
IRD_TRACE_AREA_1	
through	
IRD_TRACE_AREA_16	Indicates user defined trace areas used with *irTrace(3IRAPI)*

IRD_TRACE_AREA_AS	Indicates advanced service operations such as the TTS and ASR area
IRD_TRACE_AREA_EM	Indicates event management operations area
IRD_TRACE_AREA_IN	Indicates caller input operations area including touch-tone and speech recognition input
IRD_TRACE_AREA_PM	Indicates parameter management operations area
IRD_TRACE_AREA_RM	Indicates resource management operations area
IRD_TRACE_AREA_SE	Indicates script execution area. This includes trace entries made implicitly by Script Builder applications and through **tas(1VIS)** scripts via the **trace(3TSM)** command.
IRD_TRACE_AREA_ST	Indicates the call and application initialization and completion operations area
IRD_TRACE_AREA_TS	Indicates the telephony service operations area
IRD_TRACE_AREA_VS	Indicates the voice code and play operations area
IRD_TRACE_LEVEL_1	through **IRD_TRACE_LEVEL_8** Indicate tracing level with *irTrace(3IRAPI)*
IRD_TRANSIENT	Indicates an IRAPI process type which typically supports a single channel and *exit*'s soon after calling *irDeinit(3IRAPI)*
IRD_TRUE	Indicates a logical true
IRD_TR	Indicates the T/R telephony type
IRD_T1	Indicates the T1 (E&M) telephony type
IRD_VIRTUAL_CLASS	Indicates the telephony class including all virtual telephony types (for example, IRD_VIRT_CHAN)
IRD_VIRTUAL_CHANS	Indicates an abstract channel equipment group. Typically used as first argument to *irInitGroup(3IRAPI)* when requesting a virtual channel.
IRD_VIRTUAL	Indicates an abstract channel type that includes virtual as opposed to real (IRD_REAL) channels
IRD_VIRT_CHAN	Indicates the telephony type for virtual channels
IRD_UNDEFINED	Indicates an integer constant denoting some value as undefined
IRD_UNKNOWN_CLASS	Indicates a telephony class representing an invalid telephony type

IRD_WAIT_ALERT	Indicates a channel service state where the library is waiting for additional exchange of information with the switch before entering the IRD_RINGING state (specific to PRI)
IRD_WAIT_CLEAR	Indicates a channel service state where the library is waiting for the network to clear the prior call (specific to PRI)
IRD_WAIT_DNIS	Indicates a channel service state where the library is waiting for DNIS and ANI before going into the IRD_RINGING state (specific to ASAI)
IRD_WHOLE_WORD	Indicates WholeWord recognition

See Also

irGetIE(3IRAPI), irExec(3IRAPI)

IrDIALSTRINGS

Name

IrDIALSTRINGS — IRAPI dial strings

Synopsis

#include <**irapi.h**>

Description

Dial strings are passed to IRAPI functions to make outbound calls through *irCall(3IRAPI)* or dial digits through *irDial(3IRAPI)*. The call-control characters listed below are a subset of, and consistent with, the AT&T Core Symbols for Terminal Dialing. The type of dialing done (pulse or touch-tone) is determined via the IRP_OUTCALL_DIALTYPE parameter [see *IrPARAMETERS(4IRAPI)*].

The following characters are valid for dial strings:

[0-9], [a-d], *, #	Dual tone multi-frequency (DTMF) characters. Only 0-9 are allowed for pulse dialing.
''[a-z]''	Double quotation marks ("") surrounding alphabetic characters ([a-z]) cause them to be interpreted as touch-tone digits. The letter 'q' maps to 7 and the letter 'z' maps to 9. When in a string, the double quotes must be escaped with backslash.
-) (	These characters insert a delay equal in length to a single DTMF tone if touch-tone dialing is used; a delay equal in length to the inter-digit pulse wait period is used if pulse dialing is used. Delay causing characters are only operational with Tip/Ring (T/R) telephony interfaces. These characters are treated as **space** (see below) for other telephony types.
space	Any number of space or blank characters may be used to make the dial string more readable.

The maximum length of the dial string is dependent on the telephony type. Currently all digital interfaces have a maximum dial string length of 15 characters when using *irCall* or *irDial*. Applications that wish to pass more than 15 characters of data in an *irDial* request, should use multiple *irDial* requests. While analog interfaces allow up to 30 character dial strings, it is probably wise to use at most 15 character dial strings.

IrERRORS

Name

IrERRORS — Error definitions

Synopsis

#include <**irapi.h**>

Description

When IRAPI functions return a failure indication (typically with a return value of IRR_FAIL), they also set the global variable *irError* to indicate the cause of the error. The *irError* variable is set to one of the values in the list below.

In addition to failures detected when an activity is first started, the IRAPI library may detect errors that occur while attempting to complete a requested activity that was successfully initiated by a function that returned IRR_OK (or some other success value). In these cases, a DONE event (IRE_ANSWER_DONE, IRE_PLAY_DONE, etc.) is generated with an *event_mod1* value of IREM_ERROR and the *event_mod2* is set to one of the values in the list below.

If the following explanation of errors is not sufficient for resolving the problem, it may be useful to try reproducing the problem with tracing enabled. See *irTrace(1IRAPI)* in *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230, for further information about the trace command.

The following errors are possible:

IRER_BADRANGE — Indicates an invalid range of numbers was provided (starting value was greater than ending value)

IRER_BADSTATE — Indicates a function was called for a channel with an inappropriate library state

IRER_DRIVER_ERROR — Indicates a network interface driver call or SP driver call resulted in an error. The error may have been detected by the network, pack file, driver, input process, or library helper functions for the driver. Applications receiving this error as the result of a function failure should check *irSysError* for the error number that was reported by the driver call. Applications receiving this error as the result of a DONE event with *event_mod1* set to IREM_ERROR and *event_mod2* set to IRER_DRIVER_ERROR,

should check *event_mod3* for the error number that was reported by the driver call. Unlike IRER_SYSERROR described below, the error numbers are specific to the type of driver that is being used. The *irSysError* value should not be interpreted as being a UNIX *errno* value. Tracing or associated log messages may be helpful to establish the context in which the error is being reported.

IRER_ENDOFTABLE Indicates a function call resulted in a query beyond the end of a table

IRER_INTERNAL Indicates a function call resulted in an internal library error

IRER_INVALID Indicates that a IRAPI function was passed invalid or bad arguments

IRER_LOADERROR Indicates the *irRegister* function could not load the required extension library routines or encountered an error in trying to execute them

IRER_MAXCHAN_TIMESLOTS
Indicates a function failed since it would have required talking on more than the maximum number of timeslots for that channel

IRER_NODATA Indicates that a play or say request contained no data

IRER_NORESOURCES Indicates that the Resources for the requested service are non-existent or out of service

IRER_NOREGISTER Indicates an IRAPI function call was made prior to IRAPI process registration via *irRegister(3IRAPI)*

IRER_NOTFOUND Indicates an object required to complete a function call could not be found

IRER_OVERFLOW Indicates a function call resulted in some overflow condition for an IRAPI data element

IRER_OVERLAP Indicates an invalid overlap has been detected in two ranges of numbers that should have included one range of numbers as a proper subset of the other

IRER_PERMISSION Indicates a function was called by a process that has no permission to execute the function

IRER_PREVIOUS_INIT Indicates the forcible initialization of a channel has failed because this process already has a *cid* for this channel. This error type is only expected

to apply to MTC and related programs since *irInit* normally creates a pending request for a channel that is already owned by the same process.

IRER_REDUNDANT Indicates a function was called for which the service was already enabled. For example, calling *irStartEcho(3IRAPI)* when echo cancellation has already been started. When this error is returned, the service remains active.

IRER_RESOURCEBUSY Indicates a function was called but resources were not immediately available to serve the needs of the function. When provided as a modifier to an IRE_CALL_DONE event with an IREM_ERROR modifier, this indicates that a pending request for in-use CCA resources has timed-out.

IRER_RESOURCE_REMOVED
Indicates a service or activity was interrupted by a maintenance process that forcibly removed the resources required to perform the service or activity. This error code is provided as a modifier to a IRE_<activity>_DONE event following the IREM_ERROR modifier.

IRER_RESTRICTED Indicates resources required for the called function have been explicitly restricted from the *cid* via *irRestrictResource(3IRAPI)*

IRER_SERVICESTATE Indicates a function was called for a channel with an inappropriate service state

IRER_SHM_CORRUPTED Indicates the DEVTBL shared memory is corrupted. Stop and start the voice system.

IRER_SYSERROR Indicates a UNIX system call, driver call, or library call resulted in an error that has an associated UNIX *errno*. Applications receiving this error as the result of a function failure should check *irSysError* for the *errno* that was reported by the UNIX system call. Applications receiving this error as the result of a DONE event with *event_mod1* set to IREM_ERROR and *event_mod2* set to IRER_SYSERROR, should check *event_mod3* for the *errno* that was reported by the UNIX system call. IRER_SYSERROR differs from IRER_DRIVER_ERROR in that *irSysError* should be a valid UNIX *errno* when IRER_SYSERROR is reported. Further general information about UNIX *errno* values can be found *intro(2).*

IRER_TSBUS	Indicates a function failed since one or more channels were not on a TDM bus, or the time-slots could not be connected for the type of communication requested
IRER_UNSUPPORTED	Indicates a function call requested an operation that could not be supported on the channel telephony

See Also

IrEVENTS(4IRAPI), IrRETURNS(4IRAPI)

IrEVENTS

Name

IrEVENTS — IRAPI events

Synopsis

#include <**irapi.h**>

Description

The event structure is used to pass information describing an event from IRAPI to the calling process. It is defined as:

```
typedef struct ir_event {
    channel_id cid;
    int event_id;
    int tag;
    int event_mod1;
    int event_mod2;
    int event_mod3;
    int event_mod4;
    int event_text_len;
    void *event_text;
} ir_event_t
```

The *ir_event* modifiers are:

cid Specifies the channel identifier associated with the event. This member is always included. Set this value to IRD_NULL if not applicable.

event_id Indicates which event (IRE_*) occurred. These are symbolically defined and are listed below. This member is always included.

tag Specifies an identifier passed by the application with the asynchronous routines. Its value is defined only when an event affects a previous request that includes the value *tag*. If no tag is associated with the event, *tag* is set to -1.

event_modN Specifies up to four codes (*event_mod1* through *event_mod4*) that provide more detailed information on the event. These members are not defined for all events. Any event modifiers not used are set to IREM_NULL.

event_text_len Specifies the length of **event_text**. Set this value to 0 if it is not used.

event_text Specifies a pointer to an area of arbitrary data whose interpretation is event dependent. Set this value to IRD_NULL if it is not used.

Events are used to inform the application about completion of activities, intermediate activity progress, network conditions, or caller input. An application can disable an event by masking it. To mask an event, an application would call *irSetEvent()* with the event action set to IRF_IGNORE. Events can be used to interrupt activities by setting the event action to an OR'd list of action flags [see *irEvents(3IRAPI)*]. By default, most event actions are set to IRF_NOTIFY.

Following is a list of IRAPI events and their associated modifiers. Each event includes whether it can masked or not. Event actions are set to IRF_NOTIFY by default. Exceptions are noted where appropriate.

IRE_ANSWER_DONE This event is non-maskable. This event indicates the completion of a request to answer a call via *irAnswer(3IRAPI)*.

- **IREM_COMPLETE** This event modifier indicates that the answer completed successfully and the cid was put into a IRD_ACTIVE channel service state.
- **IREM_ERROR** This event modifier indicates the request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values. No change in channel service state occurred.

IRE_BACKGROUND This event is maskable. This event indicates the results of a request to start background play on a particular channel.

- **IREM_COMPLETE** This event modifier indicates that the background play was initiated successfully.

IREM_INTERRUPT This event modifier indicates that the background play was interrupted. This is likely due to a hardware failure or resource removal.

IREM_ERROR This event modifier indicates the request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values. No change in channel service state occurred.

IRE_BOARD_DENY This event is non-maskable. A request to remove a board from service has been denied. This event is only expected by MTC and related processes that take boards out of service. The board that has been denied is identified by *event_mod1*.

IRE_BOARD_GRANT This event is non-maskable. A request to remove a board from service has been granted. This event is only expected by MTC and related processes that take boards out of service. The board that has been granted is identified by *event_mod1*.

IRE_CALL_DONE This event is non-maskable. This event indicates the request to make an outbound call via *irCall(3IRAPI)* completed. The *tag* for this event is the *tag* provided as an argument to *irCall*. Event modifiers are identical to those used for IRE_CALL_PROG. The distinction between IRE_CALL_PROG and IRE_CALL_DONE is that IRE_CALL_DONE indicates that the *irCall* request has completed. IRE_CALL_PROG indicates intermediate events occur during the call whether a *irCall* is active or not.

IRE_CALL_PROG This event is maskable. This event occurs when a call progress tone or message has been detected. The *tag* for this event is undefined (most likely set to 0) since it is ambiguous as to what tag would apply to the event. *Event_mod1* for this event contains the specific tone or message received:

IREM_RINGBACK This modifier to the IRE_CALL_PROG event occurs for each detected ring returned by the network. An IREM_RINGBACK is currently only expected with analog (tip/ring) channels. When provided as a modifier to the IRE_CALL_DONE event, this means that the first audible ring was detected while IRP_OUTCALL_MAXRINGS was set to 0.

WARNING:
The audible ring heard by the called party may occur at a different time than the ringback provided via the network. If the called party answers the call on or before the first audible ring, it is possible that the VIS never receives a ringback from the network. Therefore, applications that depend upon receiving one or more IRE_CALL_PROG or IRE_CALL_DONE events with the IREM_RINGBACK modifier may fail to detect that the called party has answered the phone.

IREM_ERROR This event modifier occurs if there is an error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values. If the *event_id* is IRE_CALL_DONE, the *irCall* request has been aborted and the service state may have been modified. If the *event_id* is IRE_CALL_PROG, an error has been detected and call completion analysis (if enabled) most likely has been stopped. The application should take appropriate recovery steps.

IREM_DENY This event modifier indicates that the requested action failed as the necessary resources could not be obtained within the interval specified. This modifier is expected on an IRE_CALL_DONE when a pending request for CCA resources is denied after a timeout.

IREM_CCA_BUSY This event modifier indicates that CCA resources were depleted. This is slightly different than IREM_DENY in that it indicates the Resource Manager (RM) thought the CCA resources were available, but the CCA packfile thought that the SP circuit card was already fully used.

IREM_GLARE This event modifier only applies to IRE_CALL_DONE and indicates that an outbound call has failed since there is an incoming call on that channel. The recommended action is to call *irDeinit* to release the channel and then allow the Application Dispatch (AD) process to pass the call to an application to answer the incoming call.

IREM_GRANT This event modifier only applies to IRE_CALL_PROG and indicates that a pending request for CCA resources required for the call has been satisfied. The state is changing from IRS_CALL_PENDING to IRS_CALLING.

IREM_REORDER This event modifier occurs if there is a reorder tone (fast busy).

IREM_HIDRY This event modifier occurs when there is no line activity for 25 seconds after a call has been dialed.

IREM_DIALTONE This event modifier indicates dial tone is detected. For ISDN PRI, the ISDN cause value is contained in *event_mod2*, otherwise *event_mod2* is IREM_NULL.

IREM_STUTTER_DIALTONE This event modifier indicates stutter dial tone is detected.

IREM_MODEM This event modifier occurs if a modem tone is detected.

IREM_DIALDONE This event modifier occurs when dial is complete during the *irCall(3IRAPI)*.

IREM_BUSY	This event modifier occurs if a busy tone is detected. For ISDN PRI, the ISDN cause value is contained in *event_mod2*, otherwise *event_mod2* is IREM_NULL.
IREM_FAST_BUSY	This event modifier occurs if "fast" busy tone is detected. For ISDN PRI, the ISDN cause value is contained in *event_mod2*, otherwise *event_mod2* is IREM_NULL.
IREM_ANSWER	This event modifier occurs on answer detection (for example, voice energy detected).
IREM_ANSWER_SUP	This event modifier occurs on answer supervision from the switch.
IREM_TT	This event modifier applies only to IRE_CALL_DONE and indicates that a touch tone was detected (presumably indicating that the call was answered).
IREM_NOANSWER	This event modifier occurs when there is ringing, but no answer after IRP_OUTCALL_MAXRINGS. If IRP_OUTCALL_MAXRINGS is set to 0, IREM_RINGBACK occurs rather than IREM_NOANSWER.
IREM_TIMEOUT	This event modifier occurs if there is no call classification after the timeout implied by IRP_OUTCALL_MAXRINGS.
IREM_BLIND	This event modifier indicates the completion of a blind call.
IREM_USER1	This event modifier is reserved for customized applications and currently is not implemented.
IREM_USER2	This event modifier is reserved for customized applications and currently is not implemented.
IREM_PROT_PROV	This event modifier indicates that there was an ISDN protocol or provisioning error. The ISDN cause value is contained in *event_mod2*.

IREM_INVALID_DIALSTRING
This event modifier indicates that an outbound *irCall* or *irDial* failed due to an invalid dial string.

IREM_ISDN_VACANT
This event modifier indicates that there was an ISDN vacant code error. The ISDN cause value is contained in *event_mod2*.

IREM_SIT This event modifier indicates special information tones and primarily applies to using Full CCA (that is, when IRP_OUTCALL_CCALEVEL parameter is set to IRD_FULL_CCA). It can occur occasionally without the use of Full CCA as described below. *Event_mod2* can be:

IREM_RO_INTRALATA
Intralata reorder

IREM_RO_INTERLATA
Interlata jeorder

IREM_NC_INTRALATA
Intralata no circuit

IREM_NC_INTERLATA
Interlata no circuit

IREM_NC_INTL International no circuit

IREM_VC Vacant code

IREM_IC Intercept

This event modifier can also be reported on Tip/Ring channels without use of Full CCA. In that case, it represents a non-SIT intercept tone (such as the one used by most AT&T PBX's to indicate that an invalid extension was dialed).

IREM_FUTURE	Future, reserved
IREM_FF	Foreign failure (international)
IREM_OTHINTL	International other
IREM_OTHDOM	Domestic other
IREM_OTHINEF	Ineffective other
IREM_UNKNOWN	Unknown SIT type

IRE_CHAN_DENY This event is non-maskable. This event indicates the failed allocation of a channel to a process as a result of a call to *irInit(3IRAPI3)*, *irForceInit(3IRAPI)* or *irInitGroup(3IRAPI)* which returned IRR_PENDING. After receiving the event, processes should stop using the channel_id returned from one of these functions as it is invalid.

The event element *cid* is set to IRD_NULL. If the event is the result of either *irInit(3IRAPI)* or *irForceInit(3IRAPI)*, *event_mod1* is set to the channel number requested and *event_mod2* is set to IRD_INVALID. If the event is the result of *irInitGroup(3IRAPI)*, *event_mod1* is set to IRD_INVALID and *event_mod2* is set to the group number requested.

IRE_CHAN_GRANT This event is non-maskable. This event indicates a channel has been allocated successfully to a process as a result of a call to *irInit(3IRAPI3)*, *irForceInit(3IRAPI)* or *irInitGroup(3IRAPI)* which returned IRR_PENDING.

IRE_CHAN_IS_DONE This event is non-maskable. This event indicates a request to place a channel in-service via *irChanIS(3IRAPI)* completed.

IREM_COMPLETE	This event modifier indicates that the request completed successfully and the cid was put into a IRD_INACTIVE channel service state.
IREM_ERROR	This event modifier indicates the request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values.

IRE_CHAN_OOS_DONE This event is non-maskable. This event indicates a request to place a channel out-of-service via *irChanOOS(3IRAPI)* completed.

IREM_COMPLETE This event modifier indicates that the request completed successfully and the cid was put into a IRD_CHAN_OOS channel service state.

IREM_ERROR This event modifier indicates the request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values. No change in channel service state occurred.

IRE_CHAN_REMOVED This event is not maskable. This event occurs when a channel is taken away from a process. This can happen if the maintenance process (MTC) removes a channel due to a user request or a system hardware error, or when another process requests a channel. When and how channels are removed from a process may be controlled to some extent by setting the IRP_CHAN_NEGOTIATION parameter. See *irInit(3IRAPI)* and *IrPARAMETERS(4IRAPI)*. The channel number is returned as *event_mod1*. An application receiving this event should discontinue all use of the *cid* that was associated with the channel. The channel is deinitialized automatically, therefore *irDeinit()* should not be called by the application.

IRE_CHAN_REQUESTED This event is maskable. This event occurs when a process requests a channel from an application with the IRP_CHAN_NEGOTIATION set to IRD_CONDITIONAL. The process receiving the event may choose to release the channel immediately via *irDeinit(3IRAPI)*, complete the call and/or do any post call processing and release the channel, or completely ignore the event, thereby denying ownership to the requesting process.

IRE_CLOCK This event is maskable for *cid* based timers only. This event occurs every time a predesignated number of clock time units have elapsed (timeout interval). The *irStartTimer(3IRAPI)* function is used to specify the time interval between successive clock events on a given *cid*. The *irStartPTimer(3irPAI)* function is used to specify the time interval between successive clock events on a process wide basis. The subsequent IRE_CLOCK event has a valid *cid* element only for timers started with *irStartTimer(3IRAPI)* but not *irStartPTimer(3IRAPI)*. See *irTimer(3IRAPI)*.

IRE_CONVERT_DONE This event is non-maskable. This event indicates that a *irConvertAlgorithm(3IRAPI)* function completed.

IREM_COMPLETE This event modifier indicates the request completed successfully. *event_mod2* is the size, in bytes, of the converted object.

IREM_ERROR This event modifier indicates the request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values.

IRE_DEFOWNER This event is non-maskable. This event indicates that a channel has just been returned to its default owner. *event_mod1* indicates the channel number.

IRE_DEINIT_DONE This event is maskable. This event, by default, is set to IRF_IGNORE. This event indicates that a channel has been idled and released to the default owner after execution of *irDeinit(3IRAPI)*. Transient applications should wait for this event before *exit(2)*ing to give the library an opportunity to idle the channel. Note that the *cid* element in the event structure is set to IRD_NULL since there is no longer a *cid* associated with the channel. *event_mod1* contains the channel number released.

This event is not generated if the calling process never had ownership of the channel. This would be the case if a process attempted to gain ownership of a channel via irInit(3IRAPI) and received an IRR_PENDING return code, thereby placing the channel in the IRS_INIT_PENDING state. If the channel is released via irDeinit(3IRAPI) prior to receiving an IRE_CHAN_GRANT, the release is

immediate and no IRE_DEINIT_DONE event is generated.

IRE_DENY This event is non-maskable. This event indicates a request for some resource was denied after a resource request timeout. This event applies to some but not all functions returning IRR_PENDING where the pending resource was not granted after the timeout interval specified in IRP_RESOURCE_RETURNMODE. The mutually exclusive functions result in a DONE event (for example, IRE_CALL_DONE) with an event modifier of IREM_DENY rather than this separate IRE_DENY event.

IRE_DIAL_DONE This event is non-maskable. This event indicates a request to dial a string via *irDial* completed.

IREM_COMPLETE This event modifier indicates a dial request completed successfully.

IREM_ERROR This event modifier indicates the request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values.

IRE_DISCONNECT This event is non-maskable. This event occurs when disconnect is detected on a channel, but is not guaranteed to be generated for all telephony interface types. The event modifiers are a subset of those used for IRE_CALL_PROG or IREM_NULL.

By default, the event mask is set to interrupt outbound calls, and all play, say, and record activity when this event occurs [event mask set to (IRF_NOTIFY | IRF_CALLINTR | IRF_PLAYINTR | IRF_RECINTR | IRF_SAYINTR).

Since this event is not generated for all telephony interface types, an application using those telephony interface types should not depend on receiving an IRE_DISCONNECT for all cases of the remote end terminating a call. In particular, a T/R interface does not have a reliable means of detecting a disconnect that is independent of the application. Depending on the application, a disconnect on a T/R interface may be indicated by an IRE_WINK event or an IRE_CALL_PROG event with an event modifier of

IREM_DIALTONE, IREM_STUTTER_DIALTONE, IREM_REORDER, IREM_BUSY, or IREM_RINGBACK. The application must determine whether such events represent a disconnect or normal activity as part of an *irCall*, *irDial*, or *irFlash*.

IRE_DISCONNECT_DONE

This event is non-maskable. This event indicates a request to disconnect a call via *irDisconnect(3IRAPI)* completed.

IREM_COMPLETE This event modifier indicates the request to disconnect the call completed successfully. The channel service state has changed to IRD_INACTIVE.

IREM_ERROR This event modifier indicates the request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values.

IRE_ECHO_START This event is non-maskable. This event indicates the results of an attempt to start echo cancellation via *irStartEcho(3IRAPI)*.

IREM_COMPLETE This event modifier indicates the request to start the echo canceler completed successfully. The channel is ready to do speech recognition while prompting.

IREM_ERROR This event modifier indicates the request to start the echo canceler failed. *event_mod2* contains the error code and *event_mod3* may contain supplemental error code information.

IREM_DENY This event modifier indicates the request to start echo cancellation failed due to echo cancellation resources not being available in the number of milliseconds specified in IRP_RESOURCE_RETURNMODE when *irStartEcho(3IRAPI)* was executed.

IRE_ECHO_STOP This event is non-maskable. This event indicates that echo cancellation has been stopped.

IREM_COMPLETE This event modifier indicates echo cancellation has been successfully stopped. This is the normal result of stopping echo cancellation via *irStopEcho(3IRAPI)*.

IREM_ERROR This event modifier indicates echo cancellation has been stopped due to an error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values. When *event_mod2* is IRER_RESOURCE_REMOVED, it indicates that echo cancellation resources were forcibly removed by a maintenance process.

IRE_ENERGY This event is maskable. This event occurs when speech energy is detected on the channel while *irStartSpeechED* is active.

For historical reasons, it may also occur when cessation of ringing is detected by a T/R card while *irStartSpeechID* is inactive. In the future, a different event may represent cessation of ringing.

IRE_EXEC This event is not maskable. This event occurs when an application has been *irExec(3IRAPI)*'ed on a channel. *event_mod1* is the channel number. Applications are expected to *irInit(3IRAPI)* the channel after receiving this event. After successful initialization of the channel, the exec buffer may be accessed though the IRP_EXEC_BUF parameter.

IRE_EXTERNAL This event is maskable. This event occurs when another process sends an event to the IRAPI with *irPostEvent(3IRAPI)*. *event_text_len* contains the length of *event_text*. *event_text* points to the message header (*struct mbhdr*) and data passed by *irPostEvent(3IRAPI)*. See *irPostEvent(3IRAPI)* for details.

IRE_FLASH_DONE This event is non-maskable. This event indicates a request to perform a hook flash via *irFlash(3IRAPI)* completed.

IREM_COMPLETE This event modifier indicates the request completed successfully.

IREM_ERROR This event modifier indicates the request terminated due to an error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values.

IRE_GRANT This event is maskable. This event indicates a request to reserve one or more resources has been granted. This is a possible subsequent event for any function returning IRR_PENDING. See *irReserveResource(3IRAPI)*.

IRE_INPUT This event is maskable. This event occurs when any input is detected. *Event_mod1* can be:

IREM_TT For touch-tone input: *event_mod2* contains the touch-tone digit [see *IrDIALSTRINGS(4IRAPI)*]. The *tag* event structure field is not valid for IRE_INPUT with the IREM_TT modifier.

IRE_INPUT_DONE This event is maskable. This event occurs when the specified input parameters are satisfied. That is, input exists on the input queue such that a delimiter character has been received, a specified number of characters have been received, or an input timeout has occurred. Note that IRE_INPUT_DONE may or may not be associated with a IRE_INPUT event. If parameters are modified such that the current state of the input queue meets the conditions specified in the parameters, a IRE_INPUT_DONE event is generated by the next call to *irWait(3IRAPI)*. The *tag* field of the event structure has no meaning for this event. Possible values for *event_mod1* include:

IREM_INPUT_LENGTH This event modifier occurs when the number of input characters as specified in IRP_INPUT_LEN has been reached.

IREM_INPUT_DELIM This event modifier occurs when the input string delimiting character string as specified in IRP_INPUT_DELIM1 and IRP_INPUT_DELIM2 is received. *event_mod2* marks the position in the input queue where the delimiter matches the input.

IREM_TT_PRE This event modifier occurs when the IRP_TT_PRETIME interval elapses after issuing *irStartTTTimer(3IRAPI)* and no input is detected.

IREM_TT_INTER This event modifier occurs when the IRP_TT_INTERTIME interval elapses after the most recent input is received.

IREM_RECOG For speech recognition input: *event_mod2* indicates the disposition of the recognition, either IREM_COMPLETE or IREM_ERROR. If IREM_COMPLETE, *event_mod3* contains the number of characters placed on the input queue as a result of this event. If IREM_ERROR, *event_mod3* (and possibly *event_mod4*) contains the error codes for failure. [NOTE: The error codes are shifted down by one modifier from the locations described in *IrERRORS(4IRAPI)*.] When recognition input is received, the *tag* field of the event structure contains the value of the tag used with *irStartRecog(3IRAPI)*. Note that receiving this event automatically turns off the recognizer.

IREM_RECOG_PRE This event modifier occurs when the IRP_RECOG_PRETIME interval elapses after issuing *irStartRecogTimer(3IRAPI)* and no input is detected.

IRE_INTERRUPT This event is non-maskable. This event indicates that while a process was waiting in *irWait(3IRAPI),* it was interrupted by a UNIX signal or some other interrupt. This event may be used by a process to alert the process of an interrupt. For example, by setting signals, a process may force an exit from *irWait(3IRAPI)* at appropriate times. See *signal(2).*

IRE_LOOP This event is maskable. This event occurs when there is a change in loop current such that the presence of loop current is detected. This event, by default, is set to IRF_IGNORE.

IRE_NETWORK_ERROR This event is non-maskable. This event indicates an error has been detected by the telephone network or network interface card. In most cases, the application should call *irDeinit* to attempt to recover from the problem. Additional information about the error can be found in *event_mod2*. Currently this error is

expected only on PRI channels and *event_mod2* contains the ISDN cause value or the SP PRI error code.

This event might be preceded by an IRE_ANSWER_DONE, IRE_CALL_DONE, IRE_CHAN_OOS_DONE, or IRE_DISCONNECT_DONE with an event modifier of IREM_ERROR. If so, the application may have already called *irDeinit(3IRAPI)* by the time that this event is received by the application. No harm is expected if the application makes a redundant call to *irDeinit*. This likely results in a return of IRR_FAIL with *irSysError* set to IRER_BADSTATE.

IRE_NEWCALL This event is non-maskable. This event indicates a new call has come in on a channel. An application that is exec-ed by the AD process as a result of a new call sees this event as the first message after doing *irInit*. Most applications can safely ignore this IRE_NEWCALL since the application is probably written to answer the incoming call. If the IRE_NEWCALL is received after disconnecting from a call, the evnt indicates that a second incoming call has arrived. Most applications can deal with the second IRE_NEWCALL by doing a *irDeinit* on the channel to allow AD to assign the call to the proper application. If the *irDeinit* happens automatically by other means, the second IRE_NEWCALL can also be ignored.

IRE_NOLOOP This event is maskable. This event occurs when there is a change in loop current such that no loop current is detected.

IRE_NULL This event is non-maskable. No event has occurred. The IRE_NULL event is generated when *irCheck(3IRAPI)* is called with no event queued.

IRE_PLAY_DONE This event is non-maskable. This event indicates that play has completed. The amount of time played is available from *irGetVCount(3IRAPI)*. *Event_mod1* indicates the reason for the play completion as follows:

IREM_COMPLETE The play request completed successfully.

IREM_TALKOFF The play request was talked off by caller input.

IREM_STOPPED The play request was stopped via a software message such as *irStop.*

IREM_NODATA The play request specified contained no data. This event modifier indicates that voice file descriptor(s) were repositioned via *irLSeek(3IRAPI)* to the end of the file(s).

IREM_ERROR The play request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values.

IREM_DENY The request was terminated due to a timeout on waiting for play resources.

IRE_PLAY_PROG This event is maskable. By default, this event is set to IRF_IGNORE. When repeated play functions [*irPlay(3IRAPI), irFPlay(3IRAPI), irBPlay(3IRAPI)*] are queued, a single IRE_PLAY_PROG event occurs when each **intermediate** play is finished, except for the last one. The last event generates an IRE_PLAY_DONE. Access *irGetVCount(3IRAPI)* to determine the time played after each intermediate play completes.

IRE_RECORD_BUF This event is non-maskable. This event informs the application that a buffer of recorded speech has been placed in the area pointed to by *buf*, *buf* is provided from an earlier call to *irBRecord* [see *irRecord(3IRAPI)*]. *event_mod1* contains the number of bytes copied to the area pointed to by *buf*. The event tag is set to the tag used in the call to *irBRecord*.

IRE_RECORD_DONE This event is non-maskable. This event indicates that the requested voice record completed. The amount of time recorded is available from *irGetVCount(3IRAPI)*. The *event_mod1* modifiers are as follows:

IREM_COMPLETE The record request completed due to recording *count* milliseconds of speech where count was specified via the *irRecord* function.

IREM_TALKOFF The record request terminated due to caller input.

IREM_STOPPED The record request stopped via a software message resulting from *irStop.*

IREM_SILENCE The record request terminated due to silence detection.

IREM_ERROR The record request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values.

IREM_DENY The record request timed out while waiting for record resources.

IRE_RESERVE_DONE This event is non-maskable. This event indicates that a *irPhReserve* has completed.

IREM_COMPLETE The request has been completed successfully.

IREM_ERROR The request terminated due to error. *event_mod2* (and possibly *event_mod3*) contain error code values that may help resolve the problem. See *IrERRORS(4IRAPI)* for possible values.

IRE_RESOURCE_REMOVED

This event is maskable. This event occurs when a explicitly reserved dynamic resource is removed from a channel due to a maintenance process. If the resource was currently being used, the activity making use of it is stopped automatically. *event_mod1* contains the resource capability and *event_mod2* contains the implementation. See *IrRESOURCES(4IRAPI).*

This event does not occur when implicitly allocated resources are removed by a maintenance process. Instead, the application should expect to receive the associated *done* or *stop* event with event modifiers of IREM_ERROR and IRER_RESOURCE_REMOVED. (for example, see IRE_PLAY_DONE and IRE_ECHO_STOP).

IRE_REVERSE This event is maskable. This event is reserved for future use to indicate that current reversal has been detected on a channel.

IRE_SAY_DONE This event is non-maskable. This event indicates that the request for TTS was completed. The same *event_mod1* modifiers apply for IRE_SAY_DONE as

	were used for IRE_PLAY_DONE. *irGetVCount(3IRAPI)* is not available updated after this event, or at any time with TTS.
IRE_TS_DONE	This event is non-maskable. This event indicates an activity on a time slot initiated via *irTSEnd(3IRAPI)*completed. *event_mod1* contains the time slot for which the activity was started. *event_text* points to an event structure (ir_event_t) containing the IRE_<activity>_DONE event. For example, if voice play was started via *irTSEnd*, *event_text* points to an ir_event_t structure containing a IRE_PLAY_DONE event and all of its associated modifiers.
IRE_WINK	This event is maskable. Occurs when a wink is detected on a channel. This generally indicates that the network has disconnected an active call on a T/R channel.
IRE_USER[1-5]	These events are maskable. These events are, by default, set to IRF_IGNORE and may be used for future extensibility.

Caveats

The IREM_USER1 and IREM_USER2 event modifiers to IRE_CALL_PROG are used for customized tones, and may not be supported by both Intuity and Intuity Conversant. The mechanism for recognizing these customized tones is beyond the scope of IRAPI. These event modifiers provide a hook for IRAPI to be able to report the detection of them, once recognized.

IREM_USER3, IREM_USER4 or IREM_USER5 may be used for future extensibility.

See Also

IrERRORS(4IRAPI)

IrPARAMETERS

Name

IrPARAMETERS — IRAPI library parameters

Synopsis

#include <**irapi.h**>

Description

Each parameter has a set of valid values (including a default value). Parameters are named according to the following convention:

IRP_<***IRAPI Activity***>_<***Parameter Name***>

where <*Parameter Name*> is the name of the actual parameter and <*IRAPI Activity*> is the library activity to which the parameter applies. For example, recording is an IRAPI activity and output gain, and recording tone and recording algorithm are two of the applicable parameters.

Parameters vary in scope. Unless otherwise noted, parameters listed here are channel parameters that pertain to a specific channel id or *cid*. Global parameters affect all channels and applications on the voice system and are described in *irGlobalParam(3IRAPI)* and *irAPI.rc(4IRAPI)*. Channel specific parameters and Global parameters form disjointed sets.

All following parameters include their type, size and save on *irExec* status on a single line. Type is either int or char. Size is the maximum number of bytes the parameter value may occupy. Note that char type parameters may contain raw (non-ASCII) data; therefore, no assumptions about NULL character termination apply to char type parameters when setting and getting parameter values. All data is read or written according to the size of the parameter. Save on exec indicates whether the parameter value is saved after the channel is *irExec(3IRAPI)*ed to another application. The next few lines detail the parameters valid values, default value and description:

Parameter	type	size	save on exec

Valid Values — Describes the values to which this parameter may be set

Default Value — Describes the default value to which this parameter is set

Channel parameters are returned to defaults when *irDeinit(3IRAPI)* is executed for the *cid* or through the *irInitParam(3IRAPI)* or *irInitAllParams(3IRAPI)*. [See *irParams(3IRAPI)*.]

Global parameters are initialized when the voice system is started.

Description — Describes how and where the parameter is used

APPLICATION CONTROL PARAMETERS

IRP_SERVICE_NAME	char	16	YES

Valid Values — Any character string

Default Value — None. The service name is provided as argument to irExec, or the application name (as described below).

Description — This value identifies the service running on a channel id. The *irDeinit* function clears this parameter. The *irExec* functions set this parameter equal to the service name provided as an argument. If this parameter is not set when the *irInit* functions are called, these functions set this parameter equal to the application name that was provided in an earlier call to *irRegister*.

IRP_REGISTER	char	64	YES

Valid Values — Any block of 64 bytes

Default Value — All bytes in buffer are '\0'

Description — This parameter is used to pass register values between applications through *irExec(3IRAPI)*. In particular, transaction state machine (TSM) uses this parameter to pass TAS registers between applications. Register 0 uses bytes 0, 1, 2 and 3, register 1 uses bytes 4, 5, 6 and 7. This continues until register 16 which uses bytes 60, 61, 62 and 63. NOTE: 64 and only 64 bytes of data is always copied, even if the buffer is not 64 bytes long.

IRP_EXEC_BUF char 2048 YES

Valid Values Any block of 2048 bytes

Default Value There is no default for this parameter. If not explicitly set, it contains unknown values.

Description This parameter is used to pass an arbitrary data buffer between calls to *irExec(3IRAPI)*. In particular, TSM makes this buffer available when a TAS script is *irExec*d from an IRAPI application.

IRP_EXEC_BUF_LEN int 4 YES

Valid Values Any non-negative integer up to 2048 (IRD_MAXDATABUFFER)

Default Value 0

Description This value indicates the size of the exec buffer and is set by the application executing the call to *irExec(3IRAPI)*.

IRP_RESOURCE_RETURNMODE int 4 NO

Valid Values IRD_IMMEDIATE, IRD_BLOCKFOREVER, or a positive integer representing time in milliseconds.

Default Value IRD_IMMEDIATE

Description This parameter determines the behavior of any function which may implicitly allocate resources. The following settings determine the behavior of functions which cannot allocate resources to complete or initiate the requested operation:

IRD_IMMEDIATE	Causes the called function to return with a IRR_FAIL immediately
IRD_BLOCKFOREVER	Causes the called function to return IRR_PENDING. The applications should then wait on IRE_GRANT before continuing.
Any positive integer *N*	Causes the called function to return IRR_PENDING. The application should then wait for either IRE_GRANT or IRE_DENY. IRE_DENY occurs if the resource could not be granted in *N* milliseconds.

IRP_CHAN_NEGOTIATION int 4 YES

Valid Values IRD_ALLOW, IRD_CONDITIONAL

Default Value IRD_CONDITIONAL

Description This parameter determines the behavior of a channel when ownership for it is requested by another application via *irInit(3IRAPI)*.

If IRP_CHAN_NEGOTIATION is set to IRD_ALLOW and the channel is requested by another process, the channel immediately is deallocated from the current owner. The current owner receives the IRE_RESOURCE_REMOVED event and must discontinue use of the *channel_id*. Any future function calls referencing the *channel_id* fail immediately as the cid is marked as invalid.

If IRP_CHAN_NEGOTIATION is set to IRD_CONDITIONAL, the owning application receives the IRD_RESOURCE_REQUESTED event when the channel is requested by some other process. The application handles this situation depends on application implementation. The application may immediately free the channel via *irDeinit(3IRAPI)*, gracefully terminate the call and *irDeinit(3IRAPI)*, or choose to ignore the request for the channel.

Note that an application can never guarantee against having a channel deallocated, regardless of the value of IRP_CHAN_NEGOTIATION. The channel may be forcibly requested by a maintenance process.

IRP_TALKFILE int 4 YES

Valid Values Integer values from 0 to 65535

Default Value 255

Description This parameter is used to set the default talkfile used for *irPhraseReserve(3IRAPI)*.

CALLER INPUT PARAMETERS

IRP_INPUT_ERASECHAR1 char 2 NO

Valid Values Any character string

Default Value IRD_NULL

Description This parameter determines which input character erases the single character preceding it on the input queue [see *irGetInput(3IRAPI)*]. Erase characters may be one or two characters in length.

IRP_INPUT_ERASECHAR2 char 2 NO

Valid Values See IRP_INPUT_ERASECHAR1

Default Value See IRP_INPUT_ERASECHAR1

Description This parameter determines which input character erases all the characters preceding it on the input queue [see *irGetInput(3IRAPI)*]. Erase characters may be one or two characters in length.

IRP_INPUT_LEN int 4 NO

Valid Values Any positive integer

Default Value 0

Description This parameter determines the number of input characters that must be present on the input queue to cause the IRE_INPUT_DONE event to be generated.

IRP_INPUT_DELIM1 char 2 NO

Valid Values Any character string

Default Value IRD_NULL

Description This parameter determines the 1 or 2 character input delimiter. When a character sequence exists on the input queue matching this parameter, the IRE_INPUT_DONE event is generated. IRD_NULL indicates that the IRAPI does not match character delimiters.

IRP_INPUT_DELIM2	char	2	NO

Valid Values	See IRP_INPUT_DELIM1
Default Value	See IRP_INPUT_DELIM1
Description	See IRP_INPUT_DELIM1. Note that when either IRP_INPUT_DELIM1 or IRP_INPUT_DELIM2 exists on the input queue, IRE_INPUT_DONE is generated.

IRP_TT_PRETIME	int	4	NO

Valid Values	Any positive integer
Default Value	5000
Description	This parameter indicates the amount of time to wait, in milliseconds, for initial touch-tone input after execution of *irStartTTTimer(3IRAPI)*, if the input queue is empty.

IRP_TT_INTERTIME	int	4	NO

Valid Values	Any positive integer
Default Value	5000
Description	This parameter indicates the amount of time to wait, in milliseconds, for interdigit touch-tone input after execution of *irStartTTTimer(3IRAPI)* if the input queue is occupied or if input has been received since executing *irStartTTTimer(3IRAPI)*.

IRP_RECOG_PRETIME	int	4	NO

Valid Values	Any positive integer
Default Value	5000
Description	This parameter indicates the amount of time to wait, in milliseconds, for initial recognition input after execution of *irStartRecogTimer(3IRAPI)* if the input queue is empty.

VOICE PLAY PARAMETERS

IRP_PLAY_GAIN	int	4	YES

Valid Values: Any integer

Default Value: 0

Description: This parameter is specified in decibels (dB). Modification of this parameter increases or decreases the volume at which speech is played. This gain is relative to the OVOL specified through **cvis_menu(1)**. That is, the resulting output volume is OVOL plus IRP_PLAY_GAIN. The true gain range is determined by the network interface or speech processing hardware. The gain is adjusted as needed to meet the limitations of the hardware.

IRP_PLAY_SPEED	int	4	NO

Valid Values: Any integer

Default Value: 100 (no change in speed)

Description: This parameter is CURRENTLY UNIMPLEMENTED and has no effect on voice play. It specifies the speed at which speech is to be played. This value should be interpreted as a percentage of normal speed. Normal speed is 100%.

IRP_OUTPUT_BGGAIN	int	4	YES

Valid Values: Any integer

Default Value: 0

Description: This parameter specifies the gain applied to background play. Background speech is played at IRP_BACKGROUND_OVOL percent of unity gain [see *irAPI.rc(4IRAPI)*]. The speech volume then is adjusted relative to unity gain for the channel OVOL as set through *cvis_menu(1VIS)*. IRP_OUTPUT_BGGAIN, whose dimensions are dB, allows a final adjustment to this gain. The true gain range is determined by the network interface or speech processing hardware. The gain is adjusted as needed to meet the limitations of the hardware.

IRP_VOICE_TYPE int 4 YES

Valid Values *irSetParam(3IRAPI)* accepts any valid integer; however, IRD_TALK_VOICE, IRD_SP_VOICE and IRD_TDM_TS are the currently supported voice types. If IRP_VOICE_TYPE is not set to one of these values, attempts to use voice operations generate the IRER_SUPPORTED error code.

Default Value The default value depends on the system administration for the channel talk type.

Description This parameter indicates how voice is to be played or recorded on a channel. IRD_TALK_VOICE indicates that voice should be played on the card with the network connection (for example, a T/R card). IRD_SP_VOICE indicates that voice should be played on an SP card. IRD_TDM_TS may be used by channels or wishing to start a play on an allocated time slot. This parameter effects voice record as well.

VOICE RECORD PARAMETERS

IRP_RECORD_GAIN int 4 YES

Valid Values Any integer

Default Value 0

Description This parameter is specified in decibels (dB). Modification of this parameter increases or decreases the volume at which speech is recorded. This gain is relative to the IVOL specified through **cvis_menu**. The true gain range is determined by the network interface or speech processing hardware. The gain is adjusted as needed to meet the limitations of the hardware.

IRP_RECORD_GAIN has no affect when recording via the T/R driver (IRP_VOICE_TYPE is set to IRD_TALK_VOICE), but has an affect when recording on an SP over the TDM bus (IRP_VOICE_TYPE is set to IRD_SP_VOICE).

IRP_RECORD_ALGO int 4 NO

Valid Values See *IrALGORITHMS(4IRAPI)*. *irSetParam(3IRAPI)* accepts any integer values. Coding failures result for unsupported values.

Default Value IRA_A_CS32

Description This parameter determines the coding algorithm used for calls to the *irRecord(3IRAPI)* function.

IRP_RECORD_TONE int 4 NO

Valid Values IRD_ON, IRD_OFF

Default Value IRD_ON

Description Determines whether a beep tone will be sounded when the system is ready to record speech (activated by *irRecord(3IRAPI)*.

IRP_RECORD_PRETIME int 4 NO

Valid Values 0–30000 milliseconds

Default Value 5000

Description This parameter determines the amount of time a record request waits before timing out due to initial silence detection. Units are in milliseconds.

When recording more than 5 seconds of silence using the T/R driver, the IRP_RECORD_PRETIME and IRP_RECORD_INTERTIME parameters should be set to at least 3 seconds longer than the desired interval of silence. This interval allows for the 3 seconds of silence that are stripped by the T/R silence detection algorithm.

IRP_RECORD_INTERTIME int 4 NO

Valid Values 0–30000 milliseconds

Default Value 5000

Description This parameter determines the amount of time a record request waits before timing out due to interword word silence detection. Units are in milliseconds.

When recording more than 5 seconds of silence using the T/R driver, the IRP_RECORD_PRETIME and IRP_RECORD_INTERTIME parameters should be set to at least 3 seconds longer than the desired interval of silence. This interval allows for the 3 seconds of silence that are stripped by the T/R silence detection algorithm.

IRP_CREATE_MODE int 4 NO

Valid Values Any integer is accepted but only those file creation modes defined by *open(2)* produce meaningful results.

Default Value 0644

Description This parameter determines the file creation mode for the *irFRecord(3IRAPI)*function.

IRP_CREATE_FLAG int 4 NO

Valid Values Any integer is accepted but only those file creation flags defined by *open(2)* produce meaningful results.

Default Value O_WRONLY|O_CREAT|O_TRUNC

Description This parameter determines the action of *irFRecord(3IRAPI)* when opening a file to which to code.

TELEPHONY PARAMETERS

IRP_DTMF_MUTING int 4 YES

Valid Values IRD_ON, IRD_OFF

Default Value As indicated by the DIGITAL INTERFACES screen for digital interfaces (see the *Intuity CONVERSANT VIS V5.0 Operations*, 585-310-550). Always IRD_ON for Analog Interfaces (T/R).

Description This parameter determines whether DTMF muting is active.

With DTMF muting on, the outgoing speech is subject to occasional short muting. This prevents false touch tone detections that might otherwise result from speech played by the VIS being echoed back by the network and triggering the VIS touch tone detectors.

With DTMF muting off, the outgoing speech is not subject to occasional muting, but speech played by the VIS might be echoed back by the network and result in false reports of touch tones. When trying to pass DTMF tones through a bridged connection, it is generally desirable to turn off DTMF muting.

Setting IRP_DTMF_MUTING via the function *irSetParam* is unsupported on T/R channels the IRP_DTMF_MUTING parameter is always IRD_ON for T/R channels.

IRP_OUTCALL_ANSDET	int	4	YES

Valid Values: IRD_DEFAULT, IRD_OFF, IRD_ON

Default Value: IRD_DEFAULT

Description: This parameter determines the type of answer detection to be used for *irCall(3IRAPI)* and *irCCA(3IRAPI)* when Full CCA is being used. IRD_ON implies that answer detection is performed by having the SP card attempt to detect speech energy. Note: This is exclusive of the speech energy detection available with the T/R cards as described in *irSpeechED(3IRAPI)*. IRD_OFF implies that answer detection through energy detection is turned off (generally used on telephony interfaces when the more reliable answer supervision is available). IRD_DEFAULT implies that answer supervision is used when available (T1 and PRI interfaces) and answer detection is used in the remaining cases (T/R and Line Side T1 interfaces).

IRP_OUTCALL_DIALTYPE	int	4	YES

Valid Values: IRD_DIALTYPE_TT, IRD_DIALTYPE_MF (in future release), IRD_DIALTYPE_DP

Default Value: IRD_DIALTYPE_TT for T1 and other Digital Interfaces. For TR and other Analog Interfaces, the Type of Signaling from the Analog Interfaces screen is used as the default (see Chapter 6, "Switch Interface Administration," of *Intuity CONVERSANT VIS V5.0 Operations*, 585-310-550).

Description: This parameter determines the type of dial to be used for *irCall(3IRAPI)* and *irDial(3IRAPI)*.

IRP_OUTCALL_CCALEVEL	int	4	YES

Valid Values: *irSetParam(3IRAPI)* accepts any valid integer; however, only IRD_BLIND_CCA, IRD_SIMPLE_CCA or IRD_FULL_CCA are currently supported. If IRP_OUTCALL_CCALEVEL is not set to one of these values, attempts to use CCA operations generate the IRER_SUPPORTED error code.

Default Value: IRD_BLIND_CCA

Description: This parameter determines the level of CCA to be used with the *irCall(3IRAPI)* or *irCCA(3IRAPI)* functions.

IRP_OUTCALL_MAXRINGS int 4 YES

Valid Values Any non-negative integer

Default Value 5

Description This parameter determines how many rings *irCall(3IRAPI)* is to wait before reporting IRE_CALL_DONE with the IREM_NOANSWER modifier (or IREM_RINGBACK as described below).

Since the digital interfaces (T1, PRI, LST1) do not currently have ring detection capability, a timer is used for digital interfaces rather than counting rings heard on the channel. On analog T/R lines, the audible ring heard by the called party may occur at a different time than the ringback returned via the network. Therefore, this parameter value is just an approximation to the number of rings that the called party hears.

If this parameter is set to 0 on a T/R channel, the IRE_CALL_DONE has an IREM_RINGBACK modifier when audible ringing is detected (rather than the IREM_NOANSWER modifier). The IREM_RINGBACK modifier is not expected to occur on digital interfaces.

IRP_FLASH_DURATION int 4 YES

Valid Values Any non-zero positive integer

Default Value Depends on switch integration parameters set during system administration

Description This parameter indicates how long, in milliseconds, a hook flash is to take.

IRP_FLASH_TYPE int 4 YES

Valid Values IRD_FLASH_AND_WAIT or IRD_FLASH_NO_WAIT

Default Value IRD_FLASH_NO_WAIT

Description This parameter indicates whether the library, driver, and/or pack should wait after generating a flash.
IRD_FLASH_NO_WAIT can be used with all telephony types.
IRD_FLASH_AND_WAIT is only meaningful with LST1 and indicates that there should be a short wait after generating the flash. This wait interval gives the switch time to react to the flash and is generally long enough to allow the switch to prepare for digits to be dialed.

ADVANCED SPEECH SERVICES PARAMETERS

IRP_RECOG_TYPE int 4 NO

Valid Values *irSetParam(3IRAPI)* accepts any valid integer; however, only IRD_WHOLE_WORD and IRD_FLEX_WORD are supported. If a recognition function is called with some other value set, the IRER_UNSUPPORTED error is generated.

Default Value IRD_UNDEFINED

Description This parameter indicates the type of recognition to use with *irStartRecog(3IRAPI)* and *irStopRecog(IRAPI)*.

IRP_RECOG_GRAMMAR int 4 NO

Valid Values Any valid integer. The values that actually work depend on the recognition type in use. See the recognition header files provided with your recognition package for the accepted values.

Default Value IRD_UNDEFINED

Description This parameter determines the grammar used for a particular recognition type. Grammars are packfile dependent. Applications must specify grammars based on the installed packfiles.

IRP_TTS_TYPE	int	4	YES

Valid Values *irSetParam(3IRAPI)* accepts any valid integer; however, only IRD_SPTTS is supported. If a TTS function is called with some other value set, the IRER_UNSUPPORTED error is generated.

Default Value IRD_SPTTS

Description This value indicate the TTS type to be used by the application. Currently only IRD_SPTTS is supported.

IRP_ECHOCAN_TYPE	int	4	YES

Valid Values *irSetParam(3IRAPI)* accepts any valid integer; however, only IRD_SP_ECHO is supported. If an echo cancellation function is called with some other value set, the IRER_UNSUPPORTED error will be generated.

Default Value IRD_SP_ECHO

Description This value indicate the echo cancellation type to be used by the application. Currently only IRD_SP_ECHO is supported.

See Also

irParam(3IRAPI), irGlobalParam(3IRAPI), irAPI.rc(4IRAPI).

IrRESOURCES

Name

IrRESOURCES — System capabilities

Synopsis

#include <**irapi.h**>

Description

The reserve structure is used by various IRAPI resource control routines, it is defined as:

```
typedef struct ir_reserve {
    int capability;          /* Resource capability type */
    int implementation;      /* Implementation specific function */
} ir_reserve_t;
```

capability is a high level classification of resources. *implementation* is the implementation specific type which provides the resource. The following tables lists each capability and the currently supported implementations. Also included is the switch which *irReserveResource* or implicit allocation functions use to allocate resources of the appropriate type. The switch is a channel specific parameter.

IRC_CCA Call progress detection. This capability is required by the *irStartCCA(3IRAPI)* and *irCall(3IRAPI)* functions. This capability turns on the IRP_OUTCALL_CCA parameter whose value/resource pairs are defined as follows:

IRD_FULL_CCA	Intuity CONVERSANT SP resident call classification analysis

IRC_ECHOCAN Echo cancellation. This capability is required by the *irStartEcho(3IRAPI)*. This capability turns on the IRP_ECHOCAN_TYPE parameter whose value/resource pairs are defined as follows:

IRD_SP_ECHO	Intuity CONVERSANT SP resident echo cancellation

IRC_RECOG Speech recognition. This capability is required by *irStartRecog(3IRAPI)*. This capability turns on the IRP_RECOG_TYPE parameter whose value/resource pairs are defined as follows:

IRD_WHOLE_WORD
: Intuity CONVERSANT SP resident WholeWord speech recognition

IRD_FLEX_WORD
: Intuity CONVERSANT SP resident FlexWord or sub-word speech recognition

IRC_PLAY Voice play. This capability is required for any channel doing voice play on a channel with no bound play resources. It is also required by any channel doing voice play with echo cancellation on. See *irPlay(3IRAPI)* and *irEnd(3IRAPI)*. This capability turns on the IRP_VOICE_TYPE parameter whose value/resource pairs are defined as follows:

IRD_SP_VOICE
: Intuity CONVERSANT SP resident voice processing

IRC_RECORD Voice record. This capability is required for any channel doing voice record on a channel with no bound record resources. See *irRecord(3IRAPI)*. This capability turns on the IRP_VOICE_TYPE parameter whose value/resource pairs are defined as follows:

IRD_SP_VOICE
: Intuity CONVERSANT SP resident voice processing

IRC_TTS Text-to-Speech (TTS). This capability is required by *irEnd(3IRAPI)* when TTS requests have been queued via *irSay(3IRAPI)*. This capability turns on the IRP_TTS_TYPE parameter whose value/resource pairs are defined as follows:

IRD_SPTTS
: Intuity CONVERSANT SP resident TTS

The parameters listed above that are used as resource allocation types may take on values that are not listed here [see *IrPARAMETERS(4IRAPI)*]. When the parameters are set to the values that are not listed here, resource allocation is not required to support the capability of that type. This may mean that the resources to support the capability are bound to the channel. As an example, when IRP_VOICE_TYPE is set to IRD_TALK, voice play and record resources are resident on the T/R circuit card and are bound to the channel.

Results from resource queries are returned in the arrays of the following structures:

```
typedef struct ir_resource_list {
    int capability;             /* Resource capability type */
    int implementation;         /* Implementation specific function */
    int ids[IRD_MAX_RESOURCES]; /* Resource element ids */
} ir_resource_t;
```

The capability and implementation fields are the same as those for the ir_reserve_t structure.

ids is an array of hardware element identifiers which support the capability. For the case of Intuity CONVERSANT SP resources, the array elements are set to the card numbers assigned the indicated capability. The array is terminated with a value of IRD_INVALID.

IrRETURNS

Name

IrRETURNS — IRAPI return codes

Synopsis

#include <**irapi.h**>

Description

The following return values are used in the IRAPI:

IRR_FAIL	Returned when a function call fails for any reason. The return of this error implies that *irError* may be consulted for more information.
IRR_NULL	Returned when a pointer-returning function fails
IRR_OFF	Returned for various *irCheck* routines indicating an off condition
IRR_OK	Returned when a function is successful and no synchronous problem is encountered. Note that an asynchronous error can occur even if a function returns IRR_OK. Such an error is returned in a corresponding event.
IRR_ON	Returned for various *irCheck* routines indicating an on condition
IRR_PENDING	Returned by functions performing an implicit or explicit request for a resource but the resource cannot be granted at that time

IrSTATES

Name

IrSTATES — IRAPI channel library states

Synopsis

#include <**irapi.h**>

Description

Following is the list of possible library states in which a channel may be. Note that library states are independent of the channel service state, which may be IRD_ACTIVE, IRD_INACTIVE, IRD_RINGING, etc. By properly responding to events as they occur, an application rarely needs to check the value of the library state.

IRS_ANSWERING The channel is in the process of answering a call. (For Tip/Ring, this means taking the channel off-hook.)

IRS_CALLING The channel is in the process of making an outbound call.

IRS_CALL_PENDING The channel has made a request to make an outbound call but the outbound calling resources are not available. For *irCall(3IRAPI)*, this state is possible only if Full Call Classification Analysis (CCA) is being used and Full CCA resources are not yet available. This state may also be expressed as (IRS_CALLING|IRS_PENDING).

IRS_CCA_PENDING The channel has requested CCA resources but CCA resources are not yet available. This state may also be expressed as (IRS_CCA|IRS_PENDING).

IRS_DEINITING The channel is being deinitialized and the channel identifier must no longer be used.

IRS_DIALING The channel is dialing one or more digits.

IRS_DISCONNECTING The channel is disconnecting a call. (This means placing the channel on-hook for T/R type channels.)

IRS_ECHO_PENDING The channel has requested echo cancellation resources but echo cancellation resources are not yet available. This state may also be expressed as (IRS_ECHO_STARTING|IRS_PENDING).

IRS_ECHO_STARTING	The channel is starting echo cancellation.
IRS_ECHO_STOPPING	The channel is stopping echo cancellation.
IRS_FLASHING	The channel is performing a flash.
IRS_GOING_IS	The channel is in the process of going in-service.
IRS_GOING_OOS	The channel is in the process of going out-of-service.
IRS_IDLE	The channel is idle. In this state the channel may be waiting for input from telephony hardware or other processes.
IRS_INIT_PENDING	The channel ownership is pending from another process. This state may also be expressed as (IRS_INIT|IRS_PENDING).
IRS_PLAYING	The channel currently is playing speech.
IRS_PLAY_PENDING	A play request has been initiated via *irEnd* for the channel but has not yet been started due to insufficient play resources. This state may also be expressed as (IRS_PLAYING|IRS_PENDING).
IRS_PLAY_QUEUED	The channel has queued up play requests but the play requests have not yet been initiated. This state may also be expressed as (IRS_PLAYING|IRS_QUEUED).
IRS_RECOG_PENDING	The channel has requested speech recognition resources but speech recognition resources are not yet available. This state may also be expressed as (IRS_RECOG|IRS_PENDING).
IRS_RECORDING	The channel currently is recording voice.
IRS_RECORD_PENDING	The channel is waiting for resources to do voice recording. This state may also be expressed as (IRS_RECORD|IRS_PENDING).
IRS_RESERVING	The channel currently is reserving space for voice code activities.
IRS_RESOURCE_PENDING	The channel has requested resources that are not yet available. This state may also be expressed as (IRS_RESOURCE|IRS_PENDING).

IRS_SAYING	The channel currently is saying text via Text-to-Speech (TTS).
IRS_SAY_PENDING	The channel has executed *irEnd* after queuing text and is waiting to do TTS. This state may also be expressed as (IRS_SAYING|IRS_PENDING).
IRS_SAY_QUEUED	The channel has queued one or more requests to say text via TTS. This state may also be expressed as (IRS_SAYING|IRS_QUEUED).

See Also

irReserveResource(3IRAPI) and IrPARAMETERS(4IRAPI)

irAPI.rc

Name

/vs/data/irAPI.rc — IRAPI system-wide parameters

Description

The */vs/data/irAPI.rc* file (if present) can contain name/value pairs that can be used to override the default values for system-wide parameters associated with IRAPI.

Prior to Intuity CONVERSANT VIS V5.0, several of these parameters used to be placed in the */vs/data/tsm.rc* file used primarily by TSM. Now with the functionality of TSM distributed to several processes (including transient IRAPI applications), the *irAPI.rc* parameters need to be more readily accessible. Therefore, DBINIT creates the *devtbl.IRAPI_params* shared memory structure that is available to all processes. This structure includes all valid parameters for the *irAPI.rc* file. Suitable default values are provided for the parameters that are not included in the *irAPI.rc* file.

As in prior releases, it is not anticipated that the normal customer needs to modify these parameters. Customer engineers and add-on packages may need to modify them in special situations.

The following parameters can be accessed by using the *irGetGlobalParam(3IRAPI)* functions.

DNISWAIT This parameter specifies the number of tries to obtain DNIS on a Tip/Ring (T/R) channel before running a default script (if any). The default is 2.

ANALOG_LOSS_COMP This parameter specifies the amount of gain (in dB) to be added to standard gain adjustment for a bridge that includes an analog (T/R) channel. The default is 0. The range is -18 to +3 in 3 dB increments.

DIGITAL_LOSS_COMP This parameter specifies the amount of gain (in dB) to be added to standard gain adjustment for a bridge that includes only digital channels. This default is 0. The range is -15 to +6 in 3 dB increments.

BACKGROUND_OVOL This parameter specifies the adjustment to OVOL (output volume level) in percent for speech being played in the background. The default is 33. The range is 0 to 100.

VCHANS This parameter specifies the number of virtual channels to be allocated. The default is 1. The range is 1 to 96.

AD_READ_ONLY This parameter determines the permissions on the AD table. This default is 0 for read/write permission. Non-zero implies read-only permission.

TRACE_BUFFER_SIZE This parameter determines the number of trace messages in the trace buffer. The default is 4000. The range is 128 to 40000.

NON_AD_CHAN_LIST This parameter contains Comma or colon separated list of channel numbers or channel ranges for which the Application Dispatch (AD) process should not be the default channel owner. The default is null string. The maximum length is 256 bytes.

AD_CHANNEL_TABLE This parameter contains the full path file name for the channel table used AD library. The default is */vs/data/ad_channel_table*. The maximum length is 128 bytes.

AD_DNISANI_TABLE This parameter contains the full path file name for the DNIS/ANI table used by AD library. The default is */vs/data/ad_dnisani_table*. The maximum length is 128 bytes.

SPEECHDIR This paramter contains the full path directory name for the speech filesystem used by the IRAPI library. The default is */home2/vfs/talkfiles*. The maximum length is 128 bytes.

The following parameters are only used by TSM and can NOT be accessed by the *irGetGlobalParam(3IRAPI)* functions.

TR_D_RINGTONE_DISCONNECT

TR_D_BUSYTONE_DISCONNECT

TR_D_REORDERTONE_DISCONNECT

TR_D_DIALTONE_DISCONNECT

TR_D_STUTTERDT_DISCONNECT

These parameters are used only by TSM to help determine whether ring, busy, reorder, dialtone, or stutter dialtone are treated as disconnect events in TSM. The default is 0 (disabled) for TR_D_RINGTONE, TR_D_BUSYTONE, and TR_D_REORDERTONE. The default is 2 (enabled) for TR_D_DIALTONE_DISCONNECT and TR_D_STUTTERDT_DISCONNECT. When enabled (set to non-zero value), they causes TSM to treat the associated event as a disconnect unless an interrupt subroutine is provided.

DIP_TBL_SIZE This parameter is default hash table size used by TSM for looking up dynamic DIP names. The default is 32. The range is 32 to 67.

See Also

irGlobalParam(3IRAPI), IrPARAMETERS(4IRAPI).

irINDEX

PERMUTED INDEX

Sample Applications

B

What's in This Appendix

This appendix contains the following sample applications:

- chantest.c
- chantest_oc.h
- mkcall.c
- chantest_oc.c
- chantest_asr.c
- ct_asr_delay.c

The on-line source code for these IRAPI application examples is located in **/vs/examples/IRAPI**. This directory contains the and a executable shell script to link the speech required to run these application examples.

NOTE:
The application examples using ASR require that the US WholeWord Speech Recognition package is installed. This is true to both compile and execute the programs.

To compile the example programs using the supplied make file enter:

make -f example.mk all

This makes all of the application examples. You may make individual programs by substituting 'all' with the specific program name you want to make.

The example applications play recorded speech. The speech played by the applications is taken from the Feature Test Package. A tool called 'exsetup' is provided in the **/vs/examples/IRAPI** directory to link specific Feature Test Package phrases to the filenames expected by the example applications. Enter **exsetup** from the **/vs/examples/IRAPI** directory before attempting to run the example applications as follows:

/vs/examples/IRAPI/exsetup

NOTE:
The voice system must be running, you must be logged in as root, and the Feature Test Package must be installed for **exsetup** to terminate successfully.

To use these applications with incoming calls they must be defined to and assigned to a channel or ANI/DNIS.

Once the programs are successfully compiled, the services defined and assigned to channels or ANI/DNIS, and the programs executed, they are ready to take calls. Note that all programs register themselves as "chantest", so only one instance of the programs may be active at a time. Modification of the arguments to irRegister(3IRAPI) from within the source files is required to run multiple versions of the programs simultaneously.

chantest.c

The following example shows the chantest.c application. This application is a multi-channel IRAPI version of chantest.

```
static const char SCCS_ID[] ="%W%  Last Modified: %G% %U%";
/*
 ******************************
 *                            *
 *  chantest.c                *       !YABR
 *                            *
 ******************************
 *
 *  DESCRIPTION:
 *     Multi-channel irAPI version of chantest
 *
 *  USAGE:
 *     To run as single or multi-channel chantest application:
 *           chantest
 */
#include     <sys/types.h>
#include     <stdlib.h>
#include     <stdio.h>
#include     "irapi.h"

/* Defines for action upon IRE_PLAY_DONE */
#define IGNORE_WHEN_DONE   0
#define REPROMPT_WHEN_DONE 1
#define START_TIMER_WHEN_DONE     2
#define DISCONNECT_WHEN_DONE      3
#define INPUT_DONE_WHEN_DONE      4

/* Input queue defines */
#define INPUT_LEN   4

/* Defines for Chl State */
#define IDLE 0
#define BUSY 1

/* Global variables */

struct CHLDATA {
    int State;
    int RetryCount;
    int PlayDone;
    ir_event_t InputDoneEvent;
} *Chl;

int initData(int nchans)
{
    if(nchans <= 0) return(-1);
    if((Chl = (struct CHLDATA *)calloc(nchans, sizeof(struct CHLDATA))) == 0)
    {
```

```
       return(-1);
    }
    return(0);
}

void cleanup(const char *string, channel_id cid)
{
    int chan = irCid2Chan(cid);

    irPError(string);

    if ( chan == IRR_FAIL) {
       (void) irDeinit(cid);
       return;
    }

    if(Chl[chan].State != IDLE) {
       if (irDeinit(cid) < 0) {
           irPError ("Error on irDeinit");
       }
    }
    Chl[chan].State = IDLE;
}

int setEvents(channel_id cid)
{
    /* Enable events
     */
    if ( irSetEvent(cid, IRE_CALL_PROG, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_ENERGY, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_WINK, IRF_NOTIFY | IRF_PLAYINTR | IRF_CALLINTR)
       == IRR_FAIL ||
      irSetEvent(cid, IRE_DISCONNECT, IRF_NOTIFY | IRF_PLAYINTR |
       IRF_CALLINTR) == IRR_FAIL ) {
       return(-1);
    }
    return(0);
}

int setTTParams(channel_id cid)
{
    /* Set up for reading INPUT_LEN digits from input queue
     */
    if ( irSetParam(cid,IRP_INPUT_LEN,INPUT_LEN) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_PRETIME, 8000) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_INTERTIME, 5000) == IRR_FAIL ||
      irSetParamStr(cid, IRP_INPUT_DELIM1, "#") == IRR_FAIL ) {
       return(-1);
    }
    return(0);
}
```

```
void reprompt(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get chan from cid", cid);
       return;
    }

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

    if(irFlushInput(cid) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ||
      irEnd(cid, 0, 0) == IRR_FAIL ) {
       cleanup("Error in reprompt", cid);
    }
}

void play_tt(channel_id cid)
{
    int len;
    char buf[INPUT_LEN + 1];
    char *bufPtr;

    if ( (len = irGetInput(cid,buf,INPUT_LEN)) == IRR_FAIL ) {
       cleanup("irGetInput Failed in play_tt", cid);
       return;
    }
    buf[len] = 0;

    if (irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_IGNORE) == IRR_FAIL ){
       cleanup("irSetEvent failed in play_tt", cid);
       return;
    }

    for(bufPtr = &buf[0]; *bufPtr != 0; bufPtr++) {
       switch(*bufPtr) {
         case '0':
           (void) irFPlay(cid, 0, "/speech/n.0");
           break;
         case '1':
           (void) irFPlay(cid, 0, "/speech/n.1");
           break;
         case '2':
           (void) irFPlay(cid, 0, "/speech/n.2");
           break;
         case '3':
           (void) irFPlay(cid, 0, "/speech/n.3");
           break;
         case '4':
```

```
            (void) irFPlay(cid, 0, "/speech/n.4");
            break;
          case '5':
            (void) irFPlay(cid, 0, "/speech/n.5");
            break;
          case '6':
            (void) irFPlay(cid, 0, "/speech/n.6");
            break;
          case '7':
            (void) irFPlay(cid, 0, "/speech/n.7");
            break;
          case '8':
            (void) irFPlay(cid, 0, "/speech/n.8");
            break;
          case '9':
            (void) irFPlay(cid, 0, "/speech/n.9");
            break;
        }
    }
    if(strcmp(buf, "0000") == 0) {
       if (irFPlay(cid, 0, "/speech/bye") == IRR_FAIL) {
           cleanup ("Error on irFPlay", cid);
           return;
       }
       Chl[irCid2Chan(cid)].PlayDone = DISCONNECT_WHEN_DONE;
    } else {
       Chl[irCid2Chan(cid)].PlayDone = REPROMPT_WHEN_DONE;
    }
    if ( irLibState(cid) != IRS_PLAY_QUEUED ) {
       reprompt(cid);
       return;
    }
    if (irEnd(cid, 0, 0) < 0) {
       cleanup ("Error on irEnd", cid);
    }
}

void playInstr(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get channel from cid", cid);
       return;
    }

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

    if(irFlushInput(cid) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/all.rptd") == IRR_FAIL ||
```

```
      irFPlay(cid, 0, "/speech/term.4.0") == IRR_FAIL ||
      irEnd(cid, 0, 0) < 0) {
       cleanup ("Error in playInstr", cid);
    }
}

void input_done(channel_id cid, ir_event_t *evPtr)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL) {
       cleanup("Can't get channel from cid", cid);
       return;
    }

    switch(evPtr->event_mod1) {
      case IREM_INPUT_LENGTH:
      case IREM_INPUT_DELIM:
       Chl[chan].RetryCount = 0;
       play_tt(cid);
       break;
      case IREM_TT_PRE:
      case IREM_TT_INTER:
       if(Chl[chan].RetryCount++ >= 3) {

           if ( irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
             irFPlay(cid, 0, "/speech/aborted") == IRR_FAIL ||
             irFPlay(cid, 0, "/speech/bye") == IRR_FAIL ||
             irEnd(cid, 0, 0) == IRR_FAIL  ) {
              cleanup ("Error processing IRE_INPUT_DONE", cid);
              return;
           }
           Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
       } else {
           playInstr(cid);
       }
       break;
      default:
       cleanup("Unexpected IRE_INPUT_DONE event modifier", cid);
       break;
    }
}


void startChanTst(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get channel from cid", cid);
       return;
    }
```

```
    Chl[chan].PlayDone = IGNORE_WHEN_DONE;
    Chl[chan].RetryCount = 0;

    if (irFPlay(cid, 0, "/speech/begn.tst") < 0) {
       cleanup ("Error on irFPlay", cid);
       return;
    }
    playInstr(cid);
}

int main (int argc, char *argv[])
{
    ir_key_t  qid;
    ir_event_t      ev;
    channel_id      cid;
    int       chan;
    int i;

    if ((qid=irRegister("chantest")) < 0) {
       irPError ("Error on irRegister");
       exit (1);
    }

    if (initData(irNumChans(IRD_REAL)) < 0) {
       printf("Initialization failed\n");
       exit(1);
    }

    while (irWCheck(&ev) != IRR_FAIL) {
       cid = ev.cid;
       chan = irCid2Chan(cid);
       fprintf(stderr,"%s\n",irPrintEvent(&ev));
       switch (ev.event_id) {

         case IRE_EXEC:
           chan = ev.event_mod1;
           if (irInit (chan, &cid, IRD_IMMEDIATE, 0) != IRR_OK) {
              irPError ("Error on irInit");
              break;
           }

           Chl[chan].State = BUSY;

           if(setEvents(cid) < 0) {
              cleanup("Error on setEvents", cid);
              break;
           }
           if(setTTParams(cid) < 0) {
              cleanup ("Error on setTTParams", cid);
              break;
           }

           if (irAnswer(cid) == IRR_FAIL) {
```

```
      cleanup ("Error on irAnswer", cid);
    }
    break;

  case IRE_INPUT_DONE:
    if ( irLibState(cid) == IRS_PLAYING ) {
      Chl[chan].PlayDone = INPUT_DONE_WHEN_DONE;
      Chl[chan].InputDoneEvent = ev;
    } else {
      input_done(cid, &ev);
    }
    break;

  case IRE_INPUT:   /* Event is ignored */
    break;

  case IRE_ANSWER_DONE:
    if ( ev.event_mod1 != IREM_COMPLETE ) {
      cleanup("Error on irAnswer", cid);
      break;
    }
    startChanTst(cid);
    break;

  case IRE_PLAY_DONE:
    switch(Chl[chan].PlayDone) {
      case REPROMPT_WHEN_DONE:
       reprompt(cid);
       break;
      case START_TIMER_WHEN_DONE:
       if (irStartTTTimer(cid) == IRR_FAIL)
           cleanup("irStartTTTimer Failed", cid);
       break;
      case DISCONNECT_WHEN_DONE:
       if (irDisconnect(cid, 0) < 0) {
           cleanup ("Error on irDisconnect", cid);
       }
       break;
      case INPUT_DONE_WHEN_DONE:
       input_done(cid, &Chl[chan].InputDoneEvent);
       break;
      case IGNORE_WHEN_DONE:
       break;
      default:
       break;
    }
    break;

  case IRE_DISCONNECT_DONE:
    if (irDeinit(cid) < 0) {
      cleanup ("Error on irDeinit", cid);
    }
    break;
```

```
        case IRE_DISCONNECT:
        case IRE_WINK:
          if (irLibState(cid) == IRS_PLAYING) {
             Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
          } else if (irDisconnect(cid, 0) < 0) {
             cleanup ("Error on irDisconnect", cid);
          }
          break;

        default:
          break;
      }
    }
    irPError("irWCheck Failed.");
    exit(1);
}
```

chantest_oc.h

The following example shows the chantest_oc.h file. This file contains defines and the structure definitions for messages used to place out-bound chantest calls.

```
/*
 ********************************
 *                              *
 *  chantest_oc.h               *
 *                              *
 ********************************
 *
 *  DESCRIPTION:
 *     Contains defines and structure definitions for messages
 *     used to place out-bound chantest calls
 *
 */

/* Defines */
#define PH_NUM_LEN  16

/* Message passing structure */

struct CALLDATA {
    struct mbhdr header; /* message header structure required by */
                      /* irPostEventQ */
    int groupNo;
    int MaxRings;
    int CCAType;
    char Number[PH_NUM_LEN];
} ;
```

mkcall.c

The following example shows mkcall.c requesting that chantest_oc place an outbound call.

```
static const char SCCS_ID[] ="%W%  Last Modified: %G% %U%";
/*
 ******************************
 *                            *
 *  mkcall.c                  *
 *                            *
 ******************************
 *
 *  DESCRIPTION:
 *     Request that chantest_oc place an outbound call
 *
 *
 *  USAGE:
 *     mkcall  group_no  max_rings  cca_type  ph_number
 */
#include     <sys/types.h>
#include     <stdlib.h>
#include     <stdio.h>
#include     "irapi.h"
#include     "chantest_oc.h"

/* Command line usage */
#define USAGE "Usage: %s: [<group_no> <max_rings> <cca_type> <ph_number>]\n"



int main (int argc, char *argv[])
{
    ir_key_t  qid;   /* QKey for mkcall */
    ir_key_t  qkey;  /* QKey for chantest */
    struct CALLDATA callData;
    int i;

    if ((qid=irRegister("mkcall")) < 0) {
       irPError ("Error on irRegister");
       exit (1);
    }

    if ( argc == 5 ) {
       callData.groupNo = atoi(argv[1]);
       if ( callData.groupNo < 0 || callData.groupNo >= 32 ) {
           fprintf(stderr, USAGE, argv[0]);
           exit(1);
       }
       callData.MaxRings = atoi(argv[2]);
       if ( callData.MaxRings <= 0 ) {
           fprintf(stderr, USAGE, argv[0]);
           exit(1);
```

```
      }
      if ( strncmp(argv[3],"blind",strlen(argv[3])) == 0 ) {
          callData.CCAType = IRD_BLIND_CCA;
      } else if ( strncmp(argv[3],"simple",strlen(argv[3])) == 0 ) {
          callData.CCAType = IRD_SIMPLE_CCA;
      } else if ( strncmp(argv[3],"full",strlen(argv[3])) == 0 ) {
          callData.CCAType = IRD_FULL_CCA;
      } else {
          fprintf(stderr, USAGE, argv[0]);
          exit(1);
      }
      (void) strcpy(callData.Number, argv[4]);

      if((qkey = irGetQKey("chantest")) == IRR_FAIL) {
          irPError ("Error on getting QKey for chantest (is chantest running?)");
          exit (1);
       }

      if(irPostEventQ(qkey, &callData, sizeof (struct CALLDATA)) == IRR_FAIL){
          fprintf(stderr, "failed to send call request message to chantest");
          exit(1);
      }
   } else {
      fprintf(stderr,USAGE, argv[0]);
      exit(1);
   }

   printf("call request has been sent to chantest.  Not waiting for reply\n");
   exit(0);
}
```

chantest_oc.c

The following example shows the chantest_oc.c application. This is a multi-channel IRAPI version of the chantest application that has been modified to support outcalling in response to IRE_EXTERNAL messages.

```
static const char SCCS_ID[] ="%W%  Last Modified: %G% %U%";
/*
 ******************************
 *                            *
 *  chantest_oc.c             *
 *                            *
 ******************************
 *
 *  DESCRIPTION:
 *     Multi-channel irAPI version of chantest.
 *      Modified to support outcalling in response to IRE_EXTERNAL messages.
 *
 *  USAGE:
 *     To run as single or multi-channel chantest application:
 *           chantest_oc
 */
#include     <sys/types.h>
#include     <stdlib.h>
#include     <stdio.h>
#include     "irapi.h"
#include     "chantest_oc.h"

/* Defines for action upon IRE_PLAY_DONE */
#define IGNORE_WHEN_DONE         0
#define REPROMPT_WHEN_DONE       1
#define START_TIMER_WHEN_DONE    2
#define DISCONNECT_WHEN_DONE     3
#define INPUT_DONE_WHEN_DONE     4

/* Command line usage */
#define USAGE "Usage: %s"

/* Input queue defines */
#define INPUT_LEN   4

/* Defines for Chl State */
#define IDLE 0
#define BUSY 1

/* define for max number of pending calls */
#define MAX_PENDING 50

/* Global variables */

struct PENDING_CALL {
    int pending_flag;
```

```
    struct CALLDATA call_request;
} pending_calls[MAX_PENDING];

struct CHLDATA {
    int State;
    int RetryCount;
    int PlayDone;
    ir_event_t InputDoneEvent;
} *Chl;

int initData(int nchans)
{
    if(nchans <= 0) return(-1);
    if((Chl = (struct CHLDATA *)calloc(nchans, sizeof(struct CHLDATA))) == 0)
    {
       return(-1);
    }
    return(0);
}

void cleanup(const char *string, channel_id cid)
{
    int chan = irCid2Chan(cid);

    irPError(string);

    if ( chan == IRR_FAIL) {
       (void) irDeinit(cid);
       return;
    }

    if(Chl[chan].State != IDLE) {
       if (irDeinit(cid) < 0) {
           irPError ("Error on irDeinit");
       }
    }
    Chl[chan].State = IDLE;
}

int setEvents(channel_id cid)
{
    /* Enable events
     */
    if ( irSetEvent(cid, IRE_CALL_PROG, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_ENERGY, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_WINK, IRF_NOTIFY | IRF_PLAYINTR | IRF_CALLINTR)
       == IRR_FAIL ||
      irSetEvent(cid, IRE_DISCONNECT, IRF_NOTIFY | IRF_PLAYINTR |
       IRF_CALLINTR) == IRR_FAIL ) {
       return(-1);
    }
```

```
    return(0);
}

int setTTParams(channel_id cid)
{
    /* Set up for reading INPUT_LEN digits from input queue
     */
    if ( irSetParam(cid,IRP_INPUT_LEN,INPUT_LEN) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_PRETIME, 8000) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_INTERTIME, 5000) == IRR_FAIL ||
      irSetParamStr(cid, IRP_INPUT_DELIM1, "#") == IRR_FAIL ) {
       return(-1);
    }
    return(0);
}

void reprompt(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get chan from cid", cid);
       return;
    }

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

    if(irFlushInput(cid) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ||
      irEnd(cid, 0, 0) == IRR_FAIL ) {
       cleanup("Error in reprompt", cid);
    }
}

void play_tt(channel_id cid)
{
    int len;
    char buf[INPUT_LEN + 1];
    char *bufPtr;

    if ( (len = irGetInput(cid,buf,INPUT_LEN)) == IRR_FAIL ) {
       cleanup("irGetInput Failed in play_tt", cid);
       return;
    }
    buf[len] = 0;

    if (irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_IGNORE) == IRR_FAIL ){
       cleanup("irSetEvent failed in play_tt", cid);
       return;
    }
```

```
    for(bufPtr = &buf[0]; *bufPtr != 0; bufPtr++) {
       switch(*bufPtr) {
         case '0':
           (void) irFPlay(cid, 0, "/speech/n.0");
           break;
         case '1':
           (void) irFPlay(cid, 0, "/speech/n.1");
           break;
         case '2':
           (void) irFPlay(cid, 0, "/speech/n.2");
           break;
         case '3':
           (void) irFPlay(cid, 0, "/speech/n.3");
           break;
         case '4':
           (void) irFPlay(cid, 0, "/speech/n.4");
           break;
         case '5':
           (void) irFPlay(cid, 0, "/speech/n.5");
           break;
         case '6':
           (void) irFPlay(cid, 0, "/speech/n.6");
           break;
         case '7':
           (void) irFPlay(cid, 0, "/speech/n.7");
           break;
         case '8':
           (void) irFPlay(cid, 0, "/speech/n.8");
           break;
         case '9':
           (void) irFPlay(cid, 0, "/speech/n.9");
           break;
       }
    }
    if(strcmp(buf, "0000") == 0) {
       if (irFPlay(cid, 0, "/speech/bye") == IRR_FAIL) {
           cleanup ("Error on irFPlay", cid);
           return;
       }
       Chl[irCid2Chan(cid)].PlayDone = DISCONNECT_WHEN_DONE;
    } else {
       Chl[irCid2Chan(cid)].PlayDone = REPROMPT_WHEN_DONE;
    }
    if ( irLibState(cid) != IRS_PLAY_QUEUED ) {
       reprompt(cid);
       return;
    }
    if (irEnd(cid, 0, 0) < 0) {
       cleanup ("Error on irEnd", cid);
    }
}

void playInstr(channel_id cid)
```

```
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get channel from cid", cid);
       return;
    }

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

    if(irFlushInput(cid) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/all.rptd") == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/term.4.0") == IRR_FAIL ||
      irEnd(cid, 0, 0) < 0) {
       cleanup ("Error in playInstr", cid);
    }
}

void input_done(channel_id cid, ir_event_t *evPtr)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL) {
       cleanup("Can't get channel from cid", cid);
       return;
    }

    switch(evPtr->event_mod1) {
      case IREM_INPUT_LENGTH:
      case IREM_INPUT_DELIM:
       Chl[chan].RetryCount = 0;
       play_tt(cid);
       break;
      case IREM_TT_PRE:
      case IREM_TT_INTER:
       if(Chl[chan].RetryCount++ >= 3) {

           if ( irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
             irFPlay(cid, 0, "/speech/aborted") == IRR_FAIL ||
             irFPlay(cid, 0, "/speech/bye") == IRR_FAIL ||
             irEnd(cid, 0, 0) == IRR_FAIL  ) {
              cleanup ("Error processing IRE_INPUT_DONE", cid);
              return;
           }
           Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
       } else {
           playInstr(cid);
       }
       break;
      default:
```

```
      cleanup("Unexpected IRE_INPUT_DONE event modifier", cid);
      break;
    }
}


void startChanTst(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
      cleanup("Can't get channel from cid", cid);
      return;
    }

    Chl[chan].PlayDone = IGNORE_WHEN_DONE;
    Chl[chan].RetryCount = 0;

    if (irFPlay(cid, 0, "/speech/begn.tst") < 0) {
      cleanup ("Error on irFPlay", cid);
      return;
    }
    playInstr(cid);
}

int startOutcall(channel_id cid, int req_idx)
{
    struct CALLDATA *callReqP;
    int chan = irCid2Chan(cid);

    if(req_idx < 0 || req_idx > MAX_PENDING) {
      cleanup ("Invalid call request index", cid);
      return (-1);
    }
    /* Set parameters for outcalling and call.
     */
    callReqP = &pending_calls[req_idx].call_request;

    if ( irSetParam(cid, IRP_OUTCALL_DIALTYPE, IRD_DIALTYPE_TT) == IRR_FAIL ||
      irSetParam(cid, IRP_OUTCALL_MAXRINGS, callReqP->MaxRings) == IRR_FAIL ||
      irSetParam(cid, IRP_OUTCALL_CCALEVEL, callReqP->CCAType) == IRR_FAIL ||
      irCall(cid, 1, callReqP->Number) == IRR_FAIL ) {
       cleanup("Failure in startOutcall", cid);
       return(-1);
    }
    /* Release this pending_call entry */
    pending_calls[req_idx].pending_flag = IRD_FALSE;

    return(0);
}

int seizeGroup(int req_idx) {
```

```
    struct CALLDATA *callReqP;
    int ret;
    channel_id cid;

    /* Attempt to become channel owner for a channel in the equipment
    ** group identified by the request.
    */
    callReqP = &pending_calls[req_idx].call_request;

    ret = irInitGroup(callReqP->groupNo, &cid, 10000, req_idx);

    if (ret == IRR_FAIL) {
       irPError("Error on irInitGroup");
       return(-1);
    }

    if(ret == IRR_OK) {
       if ( setEvents(cid) < 0 || setTTParams(cid) < 0 ) {
           cleanup("Error setting events or parameters", cid);
           return(-1);
       }
       return(startOutcall(cid, req_idx));
    } else {
       return(0);      /* Chan request is pending */
    }
}

int main (int argc, char *argv[])
{
    ir_key_t  qid;
    ir_event_t      ev;
    channel_id      cid;
    struct CALLDATA *msgp;
    int       chan;
    int i;

    if ((qid=irRegister("chantest")) < 0) {
       irPError ("Error on irRegister");
       exit (1);
    }

    if (initData(irNumChans(IRD_REAL)) < 0) {
       printf("Initialization failed\n");
       exit(1);
    }

    if ( argc != 1 ) {
       fprintf(stderr,USAGE, argv[0]);
       exit(1);
    }

    /* indicate that there are no pending calls */
    for (i = 0; i < MAX_PENDING; i++) {
```

```
      pending_calls[i].pending_flag = IRD_FALSE;
   }

   while (irWCheck(&ev) != IRR_FAIL) {
      cid = ev.cid;
      chan = irCid2Chan(cid);
      fprintf(stderr,"%s\n",irPrintEvent(&ev));
      switch (ev.event_id) {

        case IRE_EXTERNAL:
          /* Save the outcalling request data and soft seize a
           * channel from the specified equipment group.
           */
          msgp = (struct CALLDATA *) ev.event_text;
          for (i = 0; i < MAX_PENDING; i++) {
             if(pending_calls[i].pending_flag == IRD_FALSE) {

                 pending_calls[i].pending_flag = IRD_TRUE;
                 pending_calls[i].call_request = *msgp;
                 (void) seizeGroup(i);
                 break;
             }
          }
          break;

        case IRE_EXEC:
          chan = ev.event_mod1;
          if (irInit (chan, &cid, IRD_IMMEDIATE, 0) != IRR_OK) {
             irPError ("Error on irInit");
             break;
          }

          Chl[chan].State = BUSY;

          if(setEvents(cid) < 0) {
             cleanup("Error on setEvents", cid);
             break;
          }
          if(setTTParams(cid) < 0) {
             cleanup ("Error on setTTParams", cid);
             break;
          }

          if (irAnswer(cid) == IRR_FAIL) {
             cleanup ("Error on irAnswer", cid);
          }
          break;

        case IRE_CHAN_GRANT:

          if(setEvents(cid) < 0) {
             cleanup("Error on setEvents", cid);
             break;
```

```
  }
  if(setTTParams(cid) < 0) {
     cleanup ("Error on setTTParams", cid);
     break;
  }

  (void) startOutcall(cid, ev.tag);
  break;

case IRE_CHAN_DENY:
  (void) fprintf(stderr, "Denied ownership of channel");
  if ( ev.tag >= 0 && ev.tag <= MAX_PENDING) {
      pending_calls[ev.tag].pending_flag = IRD_FALSE;
  }
  break;

case IRE_CALL_DONE:
  switch (ev.event_mod1){
    case IREM_RINGBACK:
    case IREM_ANSWER_SUP:
    case IREM_ANSWER:
    case IREM_BLIND:
     startChanTst(cid);
     break;
    case IREM_NOANSWER:
    case IREM_HIDRY:
    case IREM_REORDER:
    case IREM_BUSY:
    case IREM_ERROR:
    default:
     if (irDisconnect(cid, 0) < 0) {
         cleanup ("Error on irDisconnect", cid);
     }
     break;
  }
  break;

case IRE_INPUT_DONE:
  if ( irLibState(cid) == IRS_PLAYING ) {
     Chl[chan].PlayDone = INPUT_DONE_WHEN_DONE;
     Chl[chan].InputDoneEvent = ev;
  } else {
     input_done(cid, &ev);
  }
  break;

case IRE_INPUT:   /* Event is ignored */
  break;

case IRE_ANSWER_DONE:
  if ( ev.event_mod1 != IREM_COMPLETE ) {
     cleanup("Error on irAnswer", cid);
     break;
```

```
            }
            startChanTst(cid);
            break;

          case IRE_PLAY_DONE:
            switch(Chl[chan].PlayDone) {
              case REPROMPT_WHEN_DONE:
               reprompt(cid);
               break;
              case START_TIMER_WHEN_DONE:
               if (irStartTTTimer(cid) == IRR_FAIL)
                   cleanup("irStartTimer Failed", cid);
               break;
              case DISCONNECT_WHEN_DONE:
               if (irDisconnect(cid, 0) < 0) {
                   cleanup ("Error on irDisconnect", cid);
               }
               break;
              case INPUT_DONE_WHEN_DONE:
               input_done(cid, &Chl[chan].InputDoneEvent);
               break;
              case IGNORE_WHEN_DONE:
               break;
              default:
               break;
            }
            break;

          case IRE_DISCONNECT_DONE:
            if (irDeinit(cid) < 0) {
               cleanup ("Error on irDeinit", cid);
            }
            break;

          case IRE_DISCONNECT:
          case IRE_WINK:
            if (irLibState(cid) == IRS_PLAYING) {
               Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
            } else if (irDisconnect(cid, 0) < 0) {
               cleanup ("Error on irDisconnect", cid);
            }
            break;

          default:
            break;
        }
    }
    irPError("irWCheck Failed.");
    exit(1);
}
```

chantest_asr.c

The following shows the chantest_asr.c application. The original chantest.c application has been modified in this example to support speech recognition.

```
static const char SCCS_ID[] ="%W%  Last Modified: %G% %U%";
/*
 ******************************
 *                            *
 *  chantest_asr.c            *       !YABR
 *                            *
 ******************************
 *
 *  DESCRIPTION:
 *     Multi-channel irAPI version of chantest
 *
 *  USAGE:
 *     To run as single or multi-channel chantest application:
 *           chantest
 */
#include     <sys/types.h>
#include     <stdlib.h>
#include     <stdio.h>
#include     "irapi.h"
#include "/att/asr/grammar_hs/US.gram.h"

/* Defines for action upon IRE_PLAY_DONE */
#define IGNORE_WHEN_DONE         0
#define REPROMPT_WHEN_DONE       1
#define START_TIMER_WHEN_DONE    2
#define DISCONNECT_WHEN_DONE     3
#define INPUT_DONE_WHEN_DONE     4

/* Input queue defines */
#define INPUT_LEN   4

/* Defines for Chl State */
#define IDLE 0
#define BUSY 1

/* Global variables */

struct CHLDATA {
    int State;
    int RetryCount;
    int PlayDone;
    ir_event_t InputDoneEvent;
} *Chl;
int initData(int nchans)
{
    if(nchans <= 0) return(-1);
    if((Chl = (struct CHLDATA *)calloc(nchans, sizeof(struct CHLDATA))) == 0)
    {
```

```
       return(-1);
    }
    return(0);
}

void cleanup(const char *string, channel_id cid)
{
    int chan = irCid2Chan(cid);

    irPError(string);

    if ( chan == IRR_FAIL) {
       (void) irDeinit(cid);
       return;
    }

    if(Chl[chan].State != IDLE) {
       if (irDeinit(cid) < 0) {
           irPError ("Error on irDeinit");
       }
    }
    Chl[chan].State = IDLE;
}

int setEvents(channel_id cid)
{
    /* Enable events
     */
    if ( irSetEvent(cid, IRE_CALL_PROG, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_ENERGY, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_WINK, IRF_NOTIFY | IRF_PLAYINTR | IRF_CALLINTR)
       == IRR_FAIL ||
      irSetEvent(cid, IRE_DISCONNECT, IRF_NOTIFY | IRF_PLAYINTR |
       IRF_CALLINTR) == IRR_FAIL ) {
       return(-1);
    }
    return(0);
}
int setTTParams(channel_id cid)
{
    /* Set up for reading INPUT_LEN digits from input queue
     */
    if ( irSetParam(cid,IRP_INPUT_LEN,INPUT_LEN) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_PRETIME, 5000) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_TYPE, IRD_WHOLE_WORD) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_GRAMMAR, US_4dig) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_PRETIME, 8000) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_INTERTIME, 5000) == IRR_FAIL ||
      irSetParamStr(cid, IRP_INPUT_DELIM1, "#") == IRR_FAIL ) {
       return(-1);
    }
```

```
    return(0);
}

void reprompt(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get chan from cid", cid);
       return;
    }

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

    if(irFlushInput(cid) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ) {
       cleanup("Error in reprompt", cid);
       return;
    }
    if ( irStartRecog(cid,0) == IRR_FAIL ) {
       cleanup("irStartRecog Error", cid);
        return;
    }
    if ( irEnd(cid, 0, 0) == IRR_FAIL ) {
       cleanup("irEnd Error", cid);
    }
}
void play_tt(channel_id cid)
{
    int len;
    char buf[INPUT_LEN + 1];
    char *bufPtr;

    if ( (len = irGetInput(cid,buf,INPUT_LEN)) == IRR_FAIL ) {
       cleanup("irGetInput Failed in play_tt", cid);
       return;
    }
    buf[len] = 0;

    if (irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_IGNORE) == IRR_FAIL ){
       cleanup("irSetEvent failed in play_tt", cid);
       return;
    }

    for(bufPtr = &buf[0]; *bufPtr != 0; bufPtr++) {
       switch(*bufPtr) {
         case '0':
           (void) irFPlay(cid, 0, "/speech/n.0");
           break;
         case '1':
```

```
            (void) irFPlay(cid, 0, "/speech/n.1");
            break;
          case '2':
            (void) irFPlay(cid, 0, "/speech/n.2");
            break;
          case '3':
            (void) irFPlay(cid, 0, "/speech/n.3");
            break;
          case '4':
            (void) irFPlay(cid, 0, "/speech/n.4");
            break;
          case '5':
            (void) irFPlay(cid, 0, "/speech/n.5");
            break;
          case '6':
            (void) irFPlay(cid, 0, "/speech/n.6");
            break;
          case '7':
            (void) irFPlay(cid, 0, "/speech/n.7");
            break;
          case '8':
            (void) irFPlay(cid, 0, "/speech/n.8");
            break;
          case '9':
            (void) irFPlay(cid, 0, "/speech/n.9");
            break;
       }
    }
    if(strcmp(buf, "0000") == 0) {
       if (irFPlay(cid, 0, "/speech/bye") == IRR_FAIL) {
           cleanup ("Error on irFPlay", cid);
           return;
       }
       Chl[irCid2Chan(cid)].PlayDone = DISCONNECT_WHEN_DONE;
    } else {
       Chl[irCid2Chan(cid)].PlayDone = REPROMPT_WHEN_DONE;
    }
    if ( irLibState(cid) != IRS_PLAY_QUEUED ) {
       reprompt(cid);
       return;
    }
    if (irEnd(cid, 0, 0) < 0) {
       cleanup ("Error on irEnd", cid);
    }
}

void playInstr(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get channel from cid", cid);
       return;
```

```
    }

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

    if(irFlushInput(cid) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/all.rptd") == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/term.4.0") == IRR_FAIL ) {
        cleanup("irStartRecog Error", cid);
        return;
    }
    if ( irStartRecog(cid,0) == IRR_FAIL ) {
        cleanup("irStartRecog Error", cid);
        return;
    }
    if ( irEnd(cid, 0, 0) == IRR_FAIL ) {
       cleanup ("Error in playInstr", cid);
    }
}

void input_done(channel_id cid, ir_event_t *evPtr)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL) {
       cleanup("Can't get channel from cid", cid);
       return;
    }

    switch(evPtr->event_mod1) {
      case IREM_INPUT_LENGTH:
      case IREM_INPUT_DELIM:
      case IREM_RECOG:
       Chl[chan].RetryCount = 0;
       play_tt(cid);
       break;
      case IREM_TT_PRE:
      case IREM_TT_INTER:
      case IREM_RECOG_PRE:
       if(Chl[chan].RetryCount++ >= 3) {

           if ( irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
             irFPlay(cid, 0, "/speech/aborted") == IRR_FAIL ||
             irFPlay(cid, 0, "/speech/bye") == IRR_FAIL ||
             irEnd(cid, 0, 0) == IRR_FAIL  ) {
              cleanup ("Error processing IRE_INPUT_DONE", cid);
              return;
           }
           Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
       } else {
           playInstr(cid);
```

```
        }
        break;
       default:
        cleanup("Unexpected IRE_INPUT_DONE event modifier", cid);
        break;
    }
}


void startChanTst(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get channel from cid", cid);
       return;
    }

    Chl[chan].PlayDone = IGNORE_WHEN_DONE;
    Chl[chan].RetryCount = 0;

    if (irFPlay(cid, 0, "/speech/begn.tst") < 0) {
       cleanup ("Error on irFPlay", cid);
       return;
    }
    playInstr(cid);
}

int main (int argc, char *argv[])
{
    ir_key_t  qid;
    ir_event_t      ev;
    channel_id      cid;
    int       chan;
    int i;

    if ((qid=irRegister("chantest")) < 0) {
       irPError ("Error on irRegister");
       exit (1);
    }

    if (initData(irNumChans(IRD_REAL)) < 0) {
       printf("Initialization failed\n");
       exit(1);
    }

    while (irWCheck(&ev) != IRR_FAIL) {
       cid = ev.cid;
       chan = irCid2Chan(cid);
       fprintf(stderr,"%s\n",irPrintEvent(&ev));
       switch (ev.event_id) {

         case IRE_EXEC:
```

```
  chan = ev.event_mod1;
  if (irInit (chan, &cid, IRD_IMMEDIATE, 0) != IRR_OK) {
     irPError ("Error on irInit");
     break;
  }

  Chl[chan].State = BUSY;

  if(setEvents(cid) < 0) {
     cleanup("Error on setEvents", cid);
     break;
  }
  if(setTTParams(cid) < 0) {
     cleanup ("Error on setTTParams", cid);
     break;
  }

  if (irAnswer(cid) == IRR_FAIL) {
     cleanup ("Error on irAnswer", cid);
  }
  break;

case IRE_INPUT_DONE:
  if ( irLibState(cid) == IRS_PLAYING ) {
     Chl[chan].PlayDone = INPUT_DONE_WHEN_DONE;
     Chl[chan].InputDoneEvent = ev;
  } else {
     input_done(cid, &ev);
  }
  break;

case IRE_INPUT:   /* Event is ignored */
  if ( irCheckRecog(cid) == IRR_ON ) {
       (void) irStopRecog(cid);
       if ( irStartTTTimer(cid) == IRR_FAIL ) {
           cleanup("Can't start TT Timer", cid);
       }
   }
   break;

case IRE_ANSWER_DONE:
  if ( ev.event_mod1 != IREM_COMPLETE ) {
     cleanup("Error on irAnswer", cid);
     break;
  }
   if (irStartEcho(cid, 0) == IRR_FAIL) {
       cleanup("Error in irStartEcho", cid);
       break;
   }
  break;

 case IRE_ECHO_START:
   if ( ev.event_mod1 == IREM_ERROR ) {
```

```
            cleanup("IRE_ECHO_START reports IREM_ERROR", cid);
            break;
          }
        startChanTst(cid);
        break;

      case IRE_PLAY_DONE:
        switch(Chl[chan].PlayDone) {
          case REPROMPT_WHEN_DONE:
           reprompt(cid);
           break;
          case START_TIMER_WHEN_DONE:
           if (irCheckRecog(cid) == IRD_ON &&
             irStartRecogTimer(cid) == IRR_FAIL) {
               cleanup("irStartRecogTimer Failed", cid);
           }
           break;
          case DISCONNECT_WHEN_DONE:
           if (irDisconnect(cid, 0) < 0) {
               cleanup ("Error on irDisconnect", cid);
           }
           break;
          case INPUT_DONE_WHEN_DONE:
           input_done(cid, &Chl[chan].InputDoneEvent);
           break;
          case IGNORE_WHEN_DONE:
           break;
          default:
           break;
        }
        break;

      case IRE_DISCONNECT_DONE:
        if (irDeinit(cid) < 0) {
           cleanup ("Error on irDeinit", cid);
        }
        break;

      case IRE_DISCONNECT:
      case IRE_WINK:
        if (irLibState(cid) == IRS_PLAYING) {
           Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
        } else if (irDisconnect(cid, 0) < 0) {
           cleanup ("Error on irDisconnect", cid);
        }
        break;

      default:
        break;
    }
}
irPError("irWCheck Failed.");
exit(1);
```

```
}
```

ct_asr_delay.c

The following example shows the ct_asr_delay.c application. This chantest_asr.c application has been modified in this example to support delayed resource allocation for speech recognition.

```
static const char SCCS_ID[] ="%W%  Last Modified: %G% %U%";
/*
 ******************************
 *                            *
 *  chantest_asr.c            *       !YABR
 *                            *
 ******************************
 *
 *  DESCRIPTION:
 *     Multi-channel irAPI version of chantest
 *
 *  USAGE:
 *     To run as single or multi-channel chantest application:
 *           chantest
 */
#include     <sys/types.h>
#include     <stdlib.h>
#include     <stdio.h>
#include     "irapi.h"
#include "/att/asr/grammar_hs/US.gram.h"

/* Defines for action upon IRE_PLAY_DONE */
#define IGNORE_WHEN_DONE   0
#define REPROMPT_WHEN_DONE 1
#define START_TIMER_WHEN_DONE    2
#define DISCONNECT_WHEN_DONE     3
#define INPUT_DONE_WHEN_DONE     4

/* Input queue defines */
#define INPUT_LEN   4

/* Defines for Chl State */
#define IDLE 0
#define BUSY 1

/* Delayed Resource Allocation Timeout */
#define DELAYED_GRANT_TIMEOUT   10000

/* Global variables */

struct CHLDATA {
    int State;
    int RetryCount;
    int PlayDone;
```

```
    ir_event_t InputDoneEvent;
} *Chl;

int initData(int nchans)
{
    if(nchans <= 0) return(-1);
    if((Chl = (struct CHLDATA *)calloc(nchans, sizeof(struct CHLDATA)))
       == 0)
    {
       return(-1);
    }
    return(0);
}

void cleanup(const char *string, channel_id cid)
{
    int chan = irCid2Chan(cid);

    irPError(string);

    if ( chan == IRR_FAIL) {
       (void) irDeinit(cid);
       return;
    }

    if(Chl[chan].State != IDLE) {
       if (irDeinit(cid) < 0) {
           irPError ("Error on irDeinit");
       }
    }
    Chl[chan].State = IDLE;
}

int setEvents(channel_id cid)
{
    /* Enable events
     */
    if ( irSetEvent(cid, IRE_CALL_PROG, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_ENERGY, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_WINK, IRF_NOTIFY | IRF_PLAYINTR | IRF_CALLINTR)
       == IRR_FAIL ||
      irSetEvent(cid, IRE_DISCONNECT, IRF_NOTIFY | IRF_PLAYINTR |
       IRF_CALLINTR) == IRR_FAIL ) {
       return(-1);
    }
    return(0);
}

int setTTParams(channel_id cid)
{
    /* Set up for reading INPUT_LEN digits from input queue
```

```
     */
    if ( irSetParam(cid,IRP_INPUT_LEN,INPUT_LEN) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_PRETIME, 5000) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_TYPE, IRD_WHOLE_WORD) == IRR_FAIL ||
      irSetParam(cid, IRP_RECOG_GRAMMAR, US_4dig) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_PRETIME, 8000) == IRR_FAIL ||
      irSetParam(cid, IRP_TT_INTERTIME, 5000) == IRR_FAIL ||
      irSetParamStr(cid, IRP_INPUT_DELIM1, "#") == IRR_FAIL ) {
       return(-1);
    }
    return(0);
}

void reprompt(channel_id cid)
{
    int chan = irCid2Chan(cid);
    int ret;  /* Return value from irStartRecog */

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get chan from cid", cid);
       return;
    }

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

    if(irFlushInput(cid) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ) {
       cleanup("Error in reprompt", cid);
       return;
    }
    ret =  irStartRecog(cid,0);
    if ( ret == IRR_FAIL ) {
       cleanup("irStartRecog Error", cid);
        return;
    } else if ( ret == IRR_PENDING ) {
       return;      /* Do not start play if resources are pending */
    }
    if ( irEnd(cid, 0, 0) == IRR_FAIL ) {
       cleanup("irEnd Error", cid);
    }
}

void play_tt(channel_id cid)
{
    int len;
    char buf[INPUT_LEN + 1];
    char *bufPtr;

    if ( (len = irGetInput(cid,buf,INPUT_LEN)) == IRR_FAIL ) {
       cleanup("irGetInput Failed in play_tt", cid);
       return;
```

```
}
buf[len] = 0;

if (irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
  irSetEvent(cid, IRE_INPUT_DONE, IRF_IGNORE) == IRR_FAIL ){
   cleanup("irSetEvent failed in play_tt", cid);
   return;
}

for(bufPtr = &buf[0]; *bufPtr != 0; bufPtr++) {
   switch(*bufPtr) {
     case '0':
       (void) irFPlay(cid, 0, "/speech/n.0");
       break;
     case '1':
       (void) irFPlay(cid, 0, "/speech/n.1");
       break;
     case '2':
       (void) irFPlay(cid, 0, "/speech/n.2");
       break;
     case '3':
       (void) irFPlay(cid, 0, "/speech/n.3");
       break;
     case '4':
       (void) irFPlay(cid, 0, "/speech/n.4");
       break;
     case '5':
       (void) irFPlay(cid, 0, "/speech/n.5");
       break;
     case '6':
       (void) irFPlay(cid, 0, "/speech/n.6");
       break;
     case '7':
       (void) irFPlay(cid, 0, "/speech/n.7");
       break;
     case '8':
       (void) irFPlay(cid, 0, "/speech/n.8");
       break;
     case '9':
       (void) irFPlay(cid, 0, "/speech/n.9");
       break;
   }
}
if(strcmp(buf, "0000") == 0) {
   if (irFPlay(cid, 0, "/speech/bye") == IRR_FAIL) {
       cleanup ("Error on irFPlay", cid);
       return;
   }
   Chl[irCid2Chan(cid)].PlayDone = DISCONNECT_WHEN_DONE;
} else {
   Chl[irCid2Chan(cid)].PlayDone = REPROMPT_WHEN_DONE;
}
if ( irLibState(cid) != IRS_PLAY_QUEUED ) {
```

```
        reprompt(cid);
        return;
    }
    if (irEnd(cid, 0, 0) == IRR_FAIL) {
        cleanup ("Error on irEnd", cid);
    }
}

void playInstr(channel_id cid)
{
    int chan = irCid2Chan(cid);
    int ret;

    if ( chan == IRR_FAIL ) {
        cleanup("Can't get channel from cid", cid);
        return;
    }

    Chl[chan].PlayDone = START_TIMER_WHEN_DONE;

    if(irFlushInput(cid) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT, IRF_PLAYINTR | IRF_NOTIFY) == IRR_FAIL ||
      irSetEvent(cid, IRE_INPUT_DONE, IRF_NOTIFY) == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/ent.4.tt") == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/all.rptd") == IRR_FAIL ||
      irFPlay(cid, 0, "/speech/term.4.0") == IRR_FAIL ) {
        cleanup("irStartRecog Error", cid);
        return;
    }

    ret =  irStartRecog(cid,0);
    if ( ret == IRR_FAIL ) {
        cleanup("irStartRecog Error", cid);
        return;
    } else if ( ret == IRR_PENDING ) {
        return;
    }

    if ( irEnd(cid, 0, 0) == IRR_FAIL ) {
        cleanup ("Error in playInstr", cid);
    }
}

void input_done(channel_id cid, ir_event_t *evPtr)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL) {
        cleanup("Can't get channel from cid", cid);
        return;
    }

    switch(evPtr->event_mod1) {
```

```
      case IREM_INPUT_LENGTH:
      case IREM_INPUT_DELIM:
      case IREM_RECOG:
       Chl[chan].RetryCount = 0;
       play_tt(cid);
       break;
      case IREM_TT_PRE:
      case IREM_TT_INTER:
      case IREM_RECOG_PRE:
       if(Chl[chan].RetryCount++ >= 3) {

           if ( irSetEvent(cid, IRE_INPUT, IRF_NOTIFY) == IRR_FAIL ||
             irFPlay(cid, 0, "/speech/aborted") == IRR_FAIL ||
             irFPlay(cid, 0, "/speech/bye") == IRR_FAIL ||
             irEnd(cid, 0, 0) == IRR_FAIL  ) {
              cleanup ("Error processing IRE_INPUT_DONE", cid);
              return;
           }
           Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
       } else {
           playInstr(cid);
       }
       break;
      default:
       cleanup("Unexpected IRE_INPUT_DONE event modifier", cid);
       break;
    }
}

void startChanTst(channel_id cid)
{
    int chan = irCid2Chan(cid);

    if ( chan == IRR_FAIL ) {
       cleanup("Can't get channel from cid", cid);
       return;
    }

    Chl[chan].PlayDone = IGNORE_WHEN_DONE;
    Chl[chan].RetryCount = 0;

    if (irFPlay(cid, 0, "/speech/begn.tst") < 0) {
       cleanup ("Error on irFPlay", cid);
       return;
    }
    playInstr(cid);
}

int main (int argc, char *argv[])
{
    ir_key_t  qid;
    ir_event_t      ev;
    channel_id      cid;
```

```
int      chan;
int i;

if ((qid=irRegister("chantest")) < 0) {
   irPError ("Error on irRegister");
   exit (1);
}

if (initData(irNumChans(IRD_REAL)) < 0) {
   printf("Initialization failed\n");
   exit(1);
}

while (irWCheck(&ev) != IRR_FAIL) {
   cid = ev.cid;
   chan = irCid2Chan(cid);
   fprintf(stderr,"%s\n",irPrintEvent(&ev));
   switch (ev.event_id) {

     case IRE_EXEC:
       chan = ev.event_mod1;
       if (irInit (chan, &cid, IRD_IMMEDIATE, 0) != IRR_OK) {
          irPError ("Error on irInit");
          break;
       }

       Chl[chan].State = BUSY;

       if(setEvents(cid) < 0) {
          cleanup("Error on setEvents", cid);
          break;
       }
       if(setTTParams(cid) < 0) {
          cleanup ("Error on setTTParams", cid);
          break;
       }

       if ( irSetParam(cid, IRP_RESOURCE_RETURNMODE, DELAYED_GRANT_TIMEOUT)
         == IRR_FAIL ) {
          cleanup ("Error on irSetParam", cid);
          break;
       }

       if (irAnswer(cid) == IRR_FAIL) {
          cleanup ("Error on irAnswer", cid);
       }
       break;

     case IRE_INPUT_DONE:
       if ( irLibState(cid) == IRS_PLAYING ) {
          Chl[chan].PlayDone = INPUT_DONE_WHEN_DONE;
          Chl[chan].InputDoneEvent = ev;
       } else {
```

```
      input_done(cid, &ev);
    }
    break;

  case IRE_INPUT:   /* Event is ignored */
    if ( irCheckRecog(cid) == IRR_ON ) {
         (void) irStopRecog(cid);
         if ( irStartTTTimer(cid) == IRR_FAIL ) {
             cleanup("Can't start TT Timer", cid);
         }
     }
     break;

  case IRE_ANSWER_DONE:
    if ( ev.event_mod1 != IREM_COMPLETE ) {
       cleanup("Error on irAnswer", cid);
       break;
    }
     if (irStartEcho(cid, 0) == IRR_FAIL) {
         cleanup("Error in irStartEcho", cid);
         break;
     }
    break;

   case IRE_ECHO_START:
     if ( ev.event_mod1 == IREM_ERROR || ev.event_mod1 ==
      IREM_DENY ) {
         cleanup("IRE_ECHO_START reports IREM_ERROR/IREM_DENY", cid);
         break;
     }
    startChanTst(cid);
    break;

  case IRE_PLAY_DONE:
    switch(Chl[chan].PlayDone) {
      case REPROMPT_WHEN_DONE:
       reprompt(cid);
       break;
      case START_TIMER_WHEN_DONE:
       if (irCheckRecog(cid) == IRD_ON &&
         irStartRecogTimer(cid) == IRR_FAIL) {
           cleanup("irStartRecogTimer Failed", cid);
       }
       break;
      case DISCONNECT_WHEN_DONE:
       if (irDisconnect(cid, 0) < 0) {
           cleanup ("Error on irDisconnect", cid);
       }
       break;
      case INPUT_DONE_WHEN_DONE:
       input_done(cid, &Chl[chan].InputDoneEvent);
       break;
      case IGNORE_WHEN_DONE:
```

```
            break;
           default:
            break;
        }
        break;

      case IRE_DISCONNECT_DONE:
        if (irDeinit(cid) < 0) {
           cleanup ("Error on irDeinit", cid);
        }
        break;

      case IRE_DISCONNECT:
      case IRE_WINK:
        if (irLibState(cid) == IRS_PLAYING) {
           Chl[chan].PlayDone = DISCONNECT_WHEN_DONE;
        } else if (irDisconnect(cid, 0) < 0) {
           cleanup ("Error on irDisconnect", cid);
        }
        break;

      case IRE_GRANT:
        if ( irLibState(cid) == IRS_PLAY_QUEUED ) {
           if ( irEnd(cid, 0, 0) == IRR_FAIL ) {
               cleanup("Error on irEnd", cid);
           }
        }
        break;

      case IRE_DENY:
        /* Recognition resources denied.  Note echo cancellation and
         * play resource allocation will be communicated through an
         * an IRE_ECHO_START and IRE_PLAY_DONE event with a IREM_DENY
         * event respectively.
         */
        cleanup("Resources for recognition denied.",cid);
        break;

      default:
        break;
    }
  }
  irPError("irWCheck Failed.");
  exit(1);
}
```

Abbreviations

A

AC
Alternating current

ACD
Automatic call distributor

AD
Application Dispatch

AD-API
Application dispatch application programming interface

ADPCM
Adaptive differential pulse code modulation

ADU
Asynchronous data unit

AGL
Application generation language

ALERT
VIS Alerter process

ANI
Automatic number identification

API
Application programming interface

ARU
Alarm relay unit

ASAI
Adjunct/Switch Application Interface

ASCII
American Standard Code for Information Interchange

ASI
Analog switch integration

B

BB
Bulletin board

bps
Bits per second

BRDG
Call bridging process

BSC
Binary synchronous communication

C

CCA
Call classification analysis

CDH
Call data handler

CELP
Continuously Excited Linear Prediction

CGEN
Voice system general message class

CICS
Customer Information Control System

CMP
Companion circuit card

CMS
Call Management System

CO
Central office

CPE
Customer provided equipment or customer premise equipment

CPN
Calling party number

CPT
Call progress tones

CPU
Central processing unit

CSU
Channel service unit

CVS
Converse vector step

D

dB
Decibels

DB
Database

DBC
Database checking process

DBMS
Database management system

DC
Direct current

DCE
Data communications equipment

DCP
Digital communications protocol

DIO
Disk input and output process

DIP
Data interface process

DMA
Direct memory access

DNIS
Dialed number identification service

DSP
Digital signal processor

DTE
Data terminal equipment

DTMF
Dual tone multi-frequency

DTR
Data terminal ready

E

EBCDIC
Extended Binary Coded Decimal Interexchange Code

EIA
Electronic Industries Association

EISA
Extended Industry Standard Architecture

EMI
Electromagnetic interference

ESD
Electrostatic discharge

ESDI
Extended Serial Data Interface

ESS
Electronic Switching System

ET
Error tracker

EXTA
External alarms feature message class

F

FCC
Federal Communications Commission

FDD
Floppy disk drive

FEP
Front end processor

FFE
Form Filler Plus feature message class

FIFO
First-in-first-out processing order

foos
Facility out-of-service state

FTS
File transfer process message class

G

GEN
PRISM logger and alerter general message class

GSE
Graphical Speech Editor

GUI
Graphical user interface

H

HDD
Hard disk drive

HLLAPI
High Level Language Application Programming Interface

HOST
Host interface process message class

hwoos
Hardware out-of-service state

Hz
Hertz

I

IBM
International Business Machines

ICK
Integrity checking process message class

ID
Identification

IDE
Integrated Disk Electronics

IE
Information element

INIT
Voice system initialization message class

inserv
In-service state

IPC
Interprocess communication

IPC
Intelligent Ports Card (IPC-900)

IPCI
Integrated personal computer interface

IRAPI
Intuity Response Application Programming Interface

IRQ
Interrupt request

ISA
Industry Standard Architecture

ISDN
Integrated Services Digital Network

ISV
Independent Software Vendor

ITAC
International Technical Assistance Center

IVP4
Integrated Voice Processing card with 4 analog channels

IVP6
Integrated Voice Processing card with 6 analog channels

IVPSS
Integrated Voice Processing System Software

K

Kbps
Kilobites per second

Kbyte
Kilobyte

L

LAN
Local area network

LDB
Local database

LED
Light-emitting diode

LIFO
Last-in-first-out processing order

LN
Load number

LOG
VIS logger process message class

LST1
Line side T1

LU
Logical unit

M

manoos
Manually out-of-service state

MAP/100
Multi-Application Platform 100

MAP/100C
Multi-Application Platform 100C

MAP/40
Multi-Application Platform 40

Mbps
Megabits per second

Mbyte
Megabyte

ms
Millisecond

msec
Millisecond

MHz
Megahertz

MTC
Maintenance process

N

NCP
Network Control Program

NEBS
Network Equipment Building Standards

NEMA
National Electrical Manufacturers Association

netoos
Network out-of-service state

NFAS
Non-Facility Associated Signaling

NFS
Network file sharing

NMVT
Network Management Vector Transport

NM-API
Network Management - Application Programming Interface

nonex
Nonexistent state

NRZ
Non Return to Zero

NRZI
Non Return to Zero Inverted

O

OEM
Original equipment manufacturer

OGA
Operator generated alert

P

PBX
Private branch exchange

PC
Personal computer

PCB
Printed circuit board

PCM
Pulse code modulation

PEC
Price element code

PRI
Primary rate interface

PSTN
Public switch telephone network

PS&BM
Power supply and battery module

R

RAM
Random access memory

RECOG
Speech recognition feature message class

RDBMS
ORACLE relational database management system

REN
Ringer equivalence number

RFS
Remote file sharing

RM
Resource manager

RMB
Remote maintenance board

RTS
Request to send

S

SBC
Sub-band coding

SCCS
Switching Control Center System

SCSI
Small Computer System Interface

SDLC
Synchronous Data Link Control

SDN
Software Defined Network

SID
Station identification

SIMM
Single inline memory module

SLIP
Serial Line Interface Protocol

SNA
Systems Network Architecture

SNMP
Simple Network Management Protocol

SP
Signal processor circuit card

SPIP
Signal processor interface process

SPPLIB
Speech processing library

SQL
Structured Query Language

SR
Speech recognition

SYS
UNIX system calls message class

sysgen
System generation

T

tas
Transaction assembler

TCC
Technology Control Center

TCP/IP
Transmission control protocol/internet protocol

TDM
Time division multiplexing

TE
Terminal emulator

THR
Threshold message class

TKR
Token Ring

TLI
Transport layer interface

TLP
Transmission level plan

T/R
Tip/Ring circuit card

TRIP
Tip/Ring interface process

TSO
Technical Service Organization

TSO
Time Share Operation

TSM
Transaction state machine process

TTS
Text-to-Speech

TWIP
T1 interface process

U

UK
United Kingdom

USOC
Universal service ordering code

UVL
Unified Voice Library

V

VDC
Video display controller

VIS
Intuity CONVERSANT Voice Information System

VPC
Voice processing comarketer

VRU
Voice response unit

VROP
Voice response output process

Glossary

Numerics

3270 interface
A link between one or more Intuity CONVERSANT Voice Information System (VIS) machines and a host mainframe. In Intuity CONVERSANT VIS documentation, the 3270 interface means the link between one or more VIS machines and an IBM host mainframe.

4ESS
A large AT&T central office switch used to route calls through AT&T's telephone network.

A

ACD
See "automatic call distributor."

ADPCM
See "adaptive differential pulse code modulation."

adaptive differential pulse code modulation
A means of encoding analog voice signals into digital signals by adaptively predicting future encoded voice signals. This adaptive modulation method reduces the number of bits required to encode voice. See also "pulse code modulation."

adjunct products
Products (for example, Adjunct/Switch Application Interface) that the Intuity VIS administers via cut-through access to the inherent management capabilities of the product itself; this is in opposition to CONVERSANT VIS's ability to administer the switch directly.

Adjunct/Switch Application Interface
An optional feature package that provides an Integrated Services Digital Network-based interface between AT&T PBX's and adjunct processors.

affiliate
A business organization that AT&T controls or which with AT&T is in partnership.

alarm relay unit
A unit used in central office telecommunication arrangements that transmits warning indicators from telephone communications equipment (like the Intuity CONVERSANT VIS) to audio.

alerter
A system process that responds to patterns of events logged by the "logdaemon" process.

analog
An analog signal, such as voice or music, that varies in a continuous manner. An analog signal may be contrasted with a digital signal, which represents only discrete states.

application
Made of several components that provide an automated version of the communication between a caller and an attendant. The Intuity CONVERSANT VIS provides several methods for creating applications, including Script Builder, the Intuity Response Application Programming Interface (IRAPI), and transaction state machine (TSM) script language.

application administration
The component of the Intuity CONVERSANT VIS that provides access to the applications currently available on your system and helps you to manage and administer them.

application installation
A two-step process in which the Intuity CONVERSANT VIS invokes the TSM script assembler for the specific application name and files are moved to the appropriate directories.

application verification
A process in which the Intuity CONVERSANT VIS verifies that all the components needed by an application are complete.

ASCII
An acronym for American Standard Code for Information Interchange, a standard for data representation. ASCII code represents alphanumeric characters as binary numbers. The code includes 128 upper- and lowercase letters, numerals, and special characters. Each alphanumeric and special character has an ASCII code (binary) equivalent that is 1 byte long.

asynchronous communication
A method of data transmission in which bits or characters are sent at irregular intervals and are spaced by start and stop bits and not by time. See also “synchronous communication.”

asynchronous data unit
An electronic communications device that allows computer systems to communicate over asynchronous lines more than 50 feet in length.

AUDIX Voice Power
A complete voice-mail messaging system accessed and operated by touch-tone telephones and integrated with a switch or “Private Branch Exchange.”

automatic call distributor
A telephone system that recognizes and answers incoming calls and completes these calls based on a set of instructions contained in a database. The Automatic Call Distributor can send the call to an operator or group of operators as soon as the operator has completed a previous call or after the system has played a message to the caller.

automatic number identification
A method of identifying the calling party by automatically receiving a string of digits that identifies the calling station of a particular customer.

B

back up
The preservation of the information in a file in a different location, so that the data is not lost in the event of hardware or system failure.

backing up an application
A utility that makes an archive copy of a completed application or makes an interim copy of an application in progress. The backup copy can be restored to the VIS if the online version is damaged, or if you make revisions and wish to go back to the previous version.

barge-in
A capability provided by WholeWord speech recognition that allow callers to speak their responses to the VIS prompt and have those responses recognized before the prompt has finished playing.

batch file
A file containing one or more lines, each of which is a command executable by the UNIX shell.

binary synchronous communications
A character-oriented synchronous link protocol.

blind transfer protocol
A protocol in which a call is completed as soon as the extension is dialed, without having to wait to see if the telephone is busy or if the caller answered.

bridging
The process of connecting one telephone network connection to another telephone network connection over the Intuity CONVERSANT VIS TDM bus. Bridging decreases the processing load on the system since an active bridge does not require speech processing, database access, host activity, etc., for the transaction.

BSC
See “binary synchronous communication.”

bundle
In the context of the Enhanced File Transfer package, this term is used to denote a single file, a group of files (package), or a combination of both.

byte
A unit of storage in the computer. On many systems, a byte is 8 bits (binary digits), the equivalent of one character of text.

C

call classification analysis
An optional feature package that allows application developers to classify the disposition of originated and transferred calls.

call data event
A parameter that specifies a list of variables that are appended to a call data record at the end of each call.

call data handler process
A software process that accumulates generic call statistics and application events.

called party number
The number dialed by someone making a telephone call. It can be used by telephone switching equipment to selectively route an incoming call to a particular department or agent.

caller
The party that calls for a service, gets connected to the Intuity CONVERSANT VIS, and interacts with the system. As the Intuity CONVERSANT VIS is also capable of making outbound calls for service, the caller can also be the person who responds to those outbound calls.

call progress tones
Standard telephony sounds that indicate the status of the call. These sounds include busy, fast busy, ringback, reorder, etc.

card cage
An area within a Intuity CONVERSANT VIS platform that contains and secures all of the standard and optional circuit cards used in the system.

cartridge tape drive
A high-capacity data storage/retrieval device that can be used to transfer large amounts of information onto high-density magnetic cartridge tape based on a predetermined format. This tape can be removed from the system and stored as a backup, or used on another system.

caution
An admonishment used when there is a possibility of a service interruption or a loss of data.

CCA
See “call classification analysis.”

CDH
See “call data handler process.”

central office
An office or location in which large telecommunication machines such as telephone switches and network access facilities are maintained. These locations follow strict installation and operation requirements.

central processing unit
A component of the Intuity CONVERSANT VIS that is based on either the Multi-Application Platform 100 (MAP/100), MAP/40, or MAP/100C.

channel
See “port.”

CICS
See “Customer Information Control System.”

circuit card upgrade
A new circuit card that replaces an existing one in the platform. Usually the replacement is an updated version of the other card, and the replacement is designed to deal with technology made obsolete by industry trends or a new VIS release.

cluster controller
A bisynchronous interface that provides a means of handling remote communication processing.

command
An instruction or request given by the user to the VIS software to perform a particular function. An entire command consists of the command name and options.

CompuLert/SCCS interface
An optional feature that enables remote or console monitoring of error messages generated from the Intuity CONVERSANT VIS. CompuLert is a centralized maintenance system for monitoring minicomputers, computer mainframes, etc. The Switching Control Center System (SCCS) is similar to the CompuLert system, but is used to support 4ESS local switching systems.

configuration
The arrangement of the software and hardware of a computer system or network. The Intuity CONVERSANT VIS configuration includes either a standard or custom processor, peripheral equipment (for example, printers, modems), and software applications. Configuration also refers to the way the switch network is set up; that is, the types of products that are in the network and how those products communicate.

configuration management
The component of the VIS that allows you to manage the current configuration of voice channels, host sessions, and database connections, assign scripts to run on specific voice channels or host sessions assign functionality to SP and T1 cards, and perform various maintenance functions.

Converse Data Return (conv_data)
A Script Builder action that supports the DEFINITY call vectoring (routing) feature by enabling the switch to retain control of vector processing in the VIS environment. It supports the DEFINITY "converse" vector command to establish a two-way routing mechanism between the switch and the VIS to facilitate data passing and return.

controller circuit card
A circuit card used on a computer system that controls its basic functionality and makes the system operational. These cards are used to control magnetic peripherals, video monitors, and basic system communications.

copying an application
A utility in which information from a source application is directed into the destination application.

coresidency
The ability of two products or services to operate and interact with each other on a single hardware platform. An example of this is the use of AUDIX Voice Power along with Intuity CONVERSANT on the same VIS platform.

CPU
See "central processing unit."

crash
An interactive utility for examining the operating system core and for determining if system parameters are being exceeded.

custom speech
Unique words or phrases to be used in Intuity CONVERSANT VIS voice prompts that AT&T records for a customer on a custom basis.

custom vocabulary
A specialized package of unique words or phrased created on a per-customer basis and used by WholeWord or FlexWord speech recognition.

Customer Information Control System
Part of the operating system that manages resources for running applications (for example, IND$FILE). Note that TSO and CMS provide analogous functionality in other host environments.

D

danger
An admonishment used when there is a possibility of personal injury.

data interface process
A software process that communicates with Script Builder applications.

database
A structured set of files, records, or tables.

database field
A field used to extract values from a local database and form the structure upon which a database is built.

database table
A structure, made up of columns and rows, that holds information in a database. Database tables provide a means of storing information that changes too often to "hard-code," or permanently store, in the transaction outline.

debug
The process of locating and correcting errors in computer programs. This process is also referred to as "troubleshooting."

default
The way a computer performs a task in the absence of other instructions.

default owner
The owner of a channel when no process takes ownership of that channel. The default owner holds all idle, in-service channels. In terms of the IRAPI, this is typically the Application Dispatch process.

diagnose
The process of performing diagnostics on Tip/Ring, T1, or SP circuit cards or a bus.

dialed number identification service
A service that allows incoming calls to contain information about the telephone number for which it is destined.

directory
A type of file used to group and organize other files or directories.

DNIS
See "dialed number identification service."

DIP
See "data interface process."

display errdata
A command that displays system errors sent to the logger.

DTMF
See "dual tone multi-frequency."

dual 3270 links
A feature that provides an additional physical unit (PU) to allow a cost-effective means of connecting to two host computers. The customer can connect a VIS to two separate FEPs or to a single FEP shared by one or more host computers. Each link supports a maximum of 32 LUs.

dual tone multi-frequency
A touch tone.

dump space
An area of the disk that is fixed in size and should equal the amount of RAM on the system. The operating system "dumps" an image of core memory upon system crashes. The dump can be fetched after rebooting for analysis of what may have caused the crash.

E

editor system
A system that allows speech phrases to be displayed and edited by a user. See "Graphical Speech Editor."

Enhanced File Transfer
A feature that allows the transferring of files automatically between the Intuity CONVERSANT VIS and a synchronous host processor on a designated logical unit.

Enhanced Serial Data Interface
A software- and hardware-controlled method used to store data on magnetic peripherals.

error message
A message on the screen indicating that something is wrong and possibly suggesting how to correct it.

Error Tracker process
See "etStub."

Ethernet
A name for a local area network that uses 10BASE5 or 10BASE2 coaxial cable and InterLAN signaling techniques.

etStub
A system process that processes pre-Version 3.1 error message logging requests. These requests are transformed and passed on to the "logdaemon" process.

event
The notification given to an application when some condition occurs.

external actions
Specific tasks and interfaces controlled by Intuity CONVERSANT VIS software that allow a Script Builder application script to invoke processes and interact with other products or services. For example, a Intuity CONVERSANT VIS application script can invoke AUDIX Voice Power functionality through the used of an external action within an application script.

F

feature
A function or capability of a product or an application within the Intuity CONVERSANT VIS.

feature package
An optionally purchased package that may contain both hardware and software resources, which provides additional functionality to a standard system.

feature_tst script package
A standard CONVERSANT VIS software program that allows a VIS user to perform self-tests of critical hardware and software functionality.

field
A "slot" in a VIS window that holds one column of information in a row.

file
A collection of data treated as a basic unit of storage.

file transfer
An option that allows you to transfer files interactively or directly to and from UNIX using the File Transfer System.

filename
Alphabetic characters used to identify a particular file.

FlexWord speech recognition
A type of speech recognition based on subword technology that recognizes phonemes or parts of words of American English vocabularies. See "subword technology."

Form Filler Plus
An optional feature package that provides the capability for application scripts to record caller's responses to prompts for later transcription and review.

function key
A key, labeled F1 through F8, on your keyboard to which the Intuity CONVERSANT VIS software gives special properties for manipulating the user interface.

G

Graphical Speech Editor
A window-driven, X Windows/Motif based, graphical user interface (GUI) that can be accessed to perform different functions associated with the creation and editing of speech files to be used by VIS applications.

H

hard disk drive
A high-capacity data storage/retrieval device that is located inside a computer platform. A hard disk drive stores data on nonremovable high-density magnetic media based on a predetermined format for retrieval by the system at a later date.

hardware
The physical components of a computer system. The central processing unit, disks, tape and floppy drives, etc., are all hardware.

hardware upgrade
Replacement of one or more fundamental platform hardware components (for example, the CPU or hard disk drive), but the existing platform and other existing optional circuit cards remain.

High Level Language Applications Programming Interface (HLLAPI)
An application programming interface that allows user to write custom applications that can communicate with the host via an API.

HLLAPI
See "High Level Language Applications Programming Interface."

host computer
A computer linked to a network providing a range of services, such as database access and computation. The host computer operates in a time-sharing manner with other computers linked to it via the network.

I

iCk
The system integrity checking process.

idle channel
A channel that either has no owner or is owned by its default owner and is onhook.

IND$FILE
The standard SNA file transfer utility that runs as an application under CICS, TSO, and CMS. IND$FILE is independent of link-level protocols such as BISYNC and SDLC.

indexed table
A table that, unlike a nonindexed table, can be searched via a field name that has been indexed.

initialize
To start up the system for the first time.

Integrated Services Digital Network
A network that provides end-to-end digital connectivity to support a wide range of voice and data services.

Integrated Voice Processing circuit card
The IVP4 or IVP6 circuit card.

intelligent transfer protocol
A transfer protocol that monitors the line after dialing is complete to determine whether a busy, reorder (fast busy), or other failure has been encountered. It also recognizes when the extension is answered or if the extension is not answered after a specified number of rings.

interface
The access point of a system. With respect to the Intuity CONVERSANT VIS, the interface is designed to provide you with easy access to the software's capabilities.

interrupt
The termination of voice and/or telephony functions when some condition occurs.

Intuity Response Application Programming Interface
A library interface the provides a standard development interface for voice-telephony applications.

ipcs
A command that reports interprocess communication facilities status.

IRAPI
See "Intuity Response Application Programming Interface."

ISDN
See "Integrated Services Digital Network."

K

keyboard mapping
In emulation mode, this feature enables the keyboard to send 3270 keyboard codes to the host according to a configuration table set up during installation.

keyword spotting
A capability provided by WholeWord Speech Recognition that allows the VIS to recognize a single word in the middle of an entire phrase spoken by a caller in response to a prompt.

L

LAN
See "local area network."

library states
The state information about channel activities maintained by the IRAPI.

line side T1
A digital method of interfacing a Intuity CONVERSANT VIS to a PBX or switch using T1-related hardware and software.

listfile
An ASCII catalog that lists the contents of one or more talkfiles. Each application script is typically associated with a separate listfile. The listfile maps speech phrase strings used by application scripts into speech phrase numbers.

local area network
A data communications network in a limited geographical area. The local area network provides communications between computers and peripherals.

local database
A database residing on the Intuity CONVERSANT VIS.

logical unit
A type of SNA Network Addressable Unit.

logdaemon
System information and error logging process.

logger
See "logdaemon."

logging on/off
Entering or exiting the Intuity CONVERSANT VIS software.

LU
See "logical unit."

M

magnetic peripherals
Data storage devices that use magnetic media to store information. Such devices include hard disk drives, floppy disk drives, and cartridge tape drives.

main screen
The Intuity CONVERSANT VIS VERSION 5.0 screen from which you are able to enter System Administration or Voice System Administration.

maintenance process
A software process that runs temporary diagnostics.

Manual Configurator Program
A software program that resolves or blocks the allocation of CPU and memory resources for controlling and optional circuit cards.

masked event
An event that an application can ignore (that is, the application can ask not to be informed of the event).

master
A board that provides clock information to the TDM bus.

megabyte
A unit of memory equal to 1,048,576 bytes (1024 x 1024). It is often rounded to one million.

Microsoft
A company that manufactures software products, primarily for IBM-compatible computers.

mirroring
A method of data backup that allows all of the data transactions to the primary hard disk drive to be copied and maintained on a second identical drive in near real time. If the primary disk drive crashes or becomes disabled, all of the data stored on it (up to 1.2 billion bytes of information) is accessible on the second mirrored disk drive.

MS-DOS
A personal computer disk operating system developed by the Microsoft Corporation.

MTC
See "maintenance process."

multi-threaded application
A single process/application that controls several channels. Each thread of the application is managed explicitly. Typically this means state information for each thread is maintained and the state of the application on each channel is tracked.

N

NetView
An optional feature package that transmits high-priority (major or critical) messages to the host as Operator-Generated Alerts (OGAs) over the 3270 host link. The NetView Alarm feature package does not require a dedicated LU.

new error logging environment
A more flexible and informative environment for logging errors and status messages (introduced in CONVERSANT VIS Version 3.1). Customer applications created earlier than V3.1 that log messages require conversion to this new environment.

new operating system
The UnixWare operating system being introduced in Intuity CONVERSANT VIS V5.0.

nonindexed table
A table that may be searched only in a sequential manner and that cannot be searched via a field name.

nonmasked event
An event that must be sent to the application. Generally, an event is nonmaskable if the applicaiton would likely encounted state transition errors by trying to ignore the event.

null value
An entry containing no value. A field containing a null value is normally displayed as blank and is different from a field containing a value of zero.

O

obsolete hardware
Hardware that is no longer supported on Intuity CONVERSANT VIS V5.0.

on-line help
Messages or information that appear on the user's screen when a "function key" (F1 through F8) is pressed.

Operator Generated Alerts
System monitoring messages transmitted from the CONVERSANT VIS or other computer system to an IBM host computer that are classified as critical or major.

option
An argument used in a command line to modify program output by modifying the execution of a command. When you do not specify any options, the command will execute according to its default options.

ORACLE
A company that produces Relational Database Management software. It is also used as a generic term that identifies a database residing on a local or remote system that is created and maintained using an ORACLE RDBMS product.

P

PBX
See "private branch exchange."

PCM
See "pulse code modulation."

peripheral (device)
Equipment such as printers or terminals that is in addition to the basic processor.

permanent process
A process that starts and initializes itself before it is needed by a caller.

phoneme
A single basic sound of particular spoken language. The English language contains 40 phonemes that represent all basic sounds used with the language. As an example, the word "one" can be represented with three phonemes, "w" - "uh" - "n." Phonemes vary between languages because of guttural and nasal inflections and syllable constructs.

phrase filtering
The rejection of unrecognized speech. The WholeWord and FlexWord speech recognition packages can be programmed to reprompt the caller if the spoken response was not recognized by the VIS.

phrase tag
A string of up to 50 characters that identify the contents of a speech phrase used by an application script.

platform migration
See “platform upgrade.”

platform upgrade
The process of replacing the existing platform with a new platform.

poll
A message sent from a central controller to an individual station on a multipoint network inviting that station to send if it has any traffic to send.

polling
A network arrangement whereby a central computer asks each remote location whether they wish to send information. This arrangement enables each user or remote data terminal to transmit and receive information on shared facilities.

port
A connection or link between two devices that allows information to travel to a desired location. See “telephone network connection.”

Primary Rate Interface
An optional feature package that provides a digital interface capable both of receiving and originating telephone calls directly from/to an AT&T 4ESS switch.

private branch exchange
A private switching system, either manual or automatic, usually serving an organization, such as a business or government agency, and usually located on the customer’s premises.

processor
In Intuity CONVERSANT VIS documentation, the computer on which UnixWare and Intuity CONVERSANT VIS software runs. In general, the part of the computer system that processes the data. Also known as the “central processing unit.”

ps
A command that shows active processes. This command displays the process table and can be used to determine which processes are consuming large amounts of system resources, such as CPU time.

pseudo driver
A driver that does not control any hardware.

pulse code modulation
A digital modulation method of encoding voice signals into digital signals. See also “adaptive differential pulse code modulation.”

R

recovery
The process of using copies of the VIS software to reconstruct files that have been lost or damaged. See also “restore.”

remote database
The component of the VIS that provides access to information not currently on the VIS.

remote maintenance board
A Intuity CONVERSANT VIS board that is equipped standard on all new MAP/100 and MAP/40 platform purchases. This card, available with a built-in modem, allows remote personnel (for example, field support) to access all Intuity CONVERSANT VIS machines with a standard simplified process.

reports administration
The component of the VIS that provides access to system reports, including VIS call classification reports, call data detail reports, call data summary reports, message log reports, and traffic reports. In addition, if AUDIX Voice Power R2.1.1 is installed on your system, the reports administration component gives you access to AUDIX Voice Power reports.

restore
The process of recovering lost or damaged files by retrieving them from available backup tapes or from another disk device. See also "recovery."

restore application
A utility that replaces a damaged application or restores an older version of an application.

reuse
The concept of reusing an existing system component after a software upgrade or platform migration.

roll back
To cancel changes to a database since the point at which changes were last committed.

rollback segment
A portion of the database that records actions that should be undone under certain circumstances. Rollback segments are used to provide transaction rollback, read consistency, and recovery.

S

sar
A command that is associated with the system activity report package.

screen pop
A method of delivering a screen of information to a telephone operator at the same time a telephone call is delivered. This is accomplished by a complex chain of tasks that include identifying the calling party number, using that information to access a local or remote ORACLE database, and pulling a "form" full of information from the database using an ORACLE database utility package.

script
The set of instructions for the Intuity CONVERSANT VIS to follow during a transaction.

Script Builder
An optional software package that provides a menu-oriented interface designed to assist in the development of custom voice response applications on the VIS.

SCSI
See "Small Computer System Interface."

shared database table
A database table that is used in more than one application.

shared speech
Speech that is a part of more than one application.

shared speech pools
A parameter that allows the user of a voice application to share speech components with other applications.

Single Inline Memory Modules
A method of containing random access memory (RAM) chips on narrow circuit card strips that attach directly to sockets on the CPU circuit card. Multiple SIMMs are sometimes installed on a single CPU circuit card.

single-threaded application
An application that runs on a single voice channel.

slave
A circuit card that depends on the TDM bus for clock information.

Small Computer System Interface
A disk drive control technology in which a single SCSI adapter card plugged into a PC slot is capable of controlling as many as seven different hard disks, optical disks, tape drives, etc.

software
The set or sets of programs that instruct the computer hardware to perform a task or series of tasks — for example, UnixWare software and the Intuity CONVERSANT VIS Version 5.0 software.

software upgrade
The installation of a new version of software. The existing platform and circuit cards are kept.

source system
The system from which you are upgrading (that is, your system as it exists *before* you upgrade).

speech energy
The amount of energy in an audio signal. Literally translated, it is the output level of the sound in every phonetic utterance.

speech envelope
The linear representation of voltage on a line. It reflects the sound wave amplitude at different intervals of time. This envelope can be plotted on a graph to represent the oscillation of an audio signal between the positive and negative extremes.

speech file
A file containing an encoded speech phrase.

speech filesystem
A collection of several talkfiles. The filesystem is organized into 16-Kbyte blocks for efficient management and retrieval of talkfiles. The Intuity CONVERSANT VIS speech filesystem is not consistent with standard UNIX filesystems, and can not be referenced with standard UNIX commands such as **ls**, **cat**, etc.

speech modeling
Creating WholeWord speech recognition algorithms by collecting thousands of different speech samples of a single word and comparing them all to obtain a statistical average of the word. This average is then used by a WholeWord speech recognition program to recognize a single spoken word.

speech phrase
A continuous speech segment encoded into a digital string.

speech space
An area that contains all digitized speech used for playback in the applications loaded on the system.

standard speech
The speech package containing simple words and phrases produced by AT&T for use with an Intuity CONVERSANT VIS. This package includes digits, numbers, days of the week, and months, each spoken with initial, medial, and falling inflection. The speech is in digitized files stored on the hard disk to be used in the voice prompts played by the VIS.

standard vocabulary
A standard package of simple word speech models provided by AT&T and used for WholeWord speech recognition purposes. These phrases include the digits "zero" through "nine," "yes," "no," and "oh."

string
A contiguous sequence of characters treated as a unit. Strings are normally bounded by white spaces, tabs, or a character designated as a separator. A string value is a specified group of characters symbolized by a variable.

Structured Query Language
A standard data programming language used with data storage and data query applications.

subword technology
A method of speech recognition that recognizes phonemes or parts of words of American English vocabularies. See "whole-word technology."

switch
A software and hardware device that controls and directs voice and data traffic. A customer-based switch is known as a "private branch exchange."

switch hook
The device at the top of most telephones that is depressed when the handset is resting in the cradle (on hook). The device is raised when the handset is picked up (the telephone is off hook).

switch hook flash
A signaling technique in which the signal is originated by momentarily depressing the "switch hook."

switch interface administration
The component of the VIS that enables you to define the interaction between the VIS and switches by allowing you to establish and modify switch interface parameters and protocol options for both analog and digital interfaces.

switch network
Two or more interconnected switching systems.

synchronous communication
A method of data transmission in which bits or characters are sent at regular time intervals, rather than being spaced by start and stop bits. See also "asynchronous communication."

System 75
An advanced digital switch supporting up to 800 lines that provides voice and data communications for its users.

System 85
An advanced digital switch supporting up to 3000 lines that provides voice and data communications for its users.

system administrator
The person assigned the responsibility of monitoring all VIS software processing, performing daily system operations and preventive maintenance, and troubleshooting errors as required.

system architecture
The manner in which the Intuity CONVERSANT VIS software is structured.

system message
An event or alarm generated by either a VIS or end-user process.

system monitor
A component of the VIS in which tests are performed to verify that each incoming telephone line and its associated tip/ring or T1 card is functional. Through the "System Monitor" component, you are able to see displays of the Voice Channel and Host Session Monitors.

T

T1
A digital transmission link with a capacity of 1.544 Mbps.

table
A collection of records that are logically grouped together.

talkfile
An ASCII file that contains the speech phrase tags and phrase tag numbers for all the phrases of a specific application. The speech phrases are organized and stored in groups. Each talkfile can contain up to 65,535 phrases and the speech filesystem can contain multiple talkfiles.

target system
The system to which you are upgrading (that is, your system as you expect it to exist *after* you upgrade).

TDM
See "time-division multiplex."

telephone network connection
The point at which a telephone network connection terminates on an Intuity CONVERSANT VIS. Supported telephone connections are Tip/Ring and T1.

Terminal Emulator
Software that allows the VIS to temporarily transform itself into a "look alike" of an IBM 3270 terminal. In addition to providing full 3270 functionality, the Terminal Emulator enables you to transfer files to and from UNIX.

Text-to-Speech
An optional feature that allows an application to play speech directly from ASCII text by converting that text to synthesized speech. The text can be used for prompts or for text retrieved from a database or host, and can be spoken in an application with prerecorded speech. Text-to-Speech application development is supported through Script Builder.

ThickNet
A 10-millimeter (10BASE5) coaxial cable used to provide InterLAN communications.

ThinNet
A 5-millimeter (10BASE2) coaxial cable used to provide InterLAN communications.

time-division multiplex
A method of serving a number of simultaneous channels over a common transmission path by assigning the transmission path sequentially to the channels, with each assignment being for a discrete time interval.

Tip/Ring
A term used to denote analog telecommunications using four-wire media.

Token/Ring
A ring type of local area network that allows any station in the network to communicate with any other station.

trace
A command that can be used to monitor the execution of a script.

traffic
The flow of information or messages through a communications network for voice, data, or audio services.

transaction
Comprised of the exchanges between the caller and the voice system. A transaction can involve one or more telephone network connections and voice responses from the Intuity CONVERSANT VIS. It can also involve one or more of the VIS optional features, such as speech recognition, 3270 host interface, FAX response, etc.

transaction state machine process
A multi-channel IRAPI application that runs applications driven by script information.

transient process
A process that is created dynamically only when needed.

troubleshoot
The process of locating and correcting errors in computer programs. This process is also referred to as debugging.

TSM
See “transaction state machine process.”

TTS
See “Text-to-Speech.”

U

UNIX Operating System
A multiuser, multitasking computer operating system developed by the Bell Telephone Laboratories division of AT&T.

UNIX shell
The command language that provides a user interface to the UNIX operating system.

upgrade image tape
A tape, optionally provided to you by the Technical Service Organization, containing the new operating system and Intuity CONVERSANT VIS V5.0 base software in a standard configuration which is compatible with your target system.

upgrade scenario
The particular combination of current hardware, software, application and target hardware, software, applications, etc.

V

vi editor
A screen editor used by the Intuity CONVERSANT VIS to create and change electronic files.

virtual channel
A channel that is not associated with an interface to the telephone network (Tip/Ring, T1, or PRI). Virtual channels are intended to run "data only" applications which do not interact with callers but may interact with DIPs. Voice or network functions (for example, coding or playing speech, call answer, origination, or transfer) will not work on a virtual channel. Virtual channel applications may be initiated only by a "virtual seizure" request to TSM from a DIP.

VIS
See "Voice Information System."

vocabulary
A collection of words that a VIS is able to recognize using either WholeWord or FlexWord speech recognition.

vocabulary activation
The set of active vocabularies that define the words and wordlists known to the FlexWord recognizer.

vocabulary loading
The process of copying the vocabulary from the system where it was developed and adding it to the target system.

voice channel
A channel that is associated with an interface to the telephone network (Tip/Ring, T1, or PRI). Any Intuity CONVERSANT VIS application can run on a voice channel. Voice channel applications may be initiated by being assigned to particular voice channels or dialed numbers to handle incoming calls or by a "soft seizure" request to TSM from a data interface process (DIP) or the **soft_szr** command.

Voice Information System
A computer connected to a telephone network that handles touch-tone input, voice response, and line transfer. The Voice Information System uses a screen-based, menu-driven user interface to interact with the system operator or administrator.

voice processing co-marketer
A company licensed to purchase voice processing equipment, such as the Intuity CONVERSANT VIS, to market and sell based on their own marketing strategies.

voice response output process
A software process that transfers digitized speech between system hardware (for example, Tip/Ring and SP cards) and data storage devices (that is, hard disk, etc.)

Voice System Administration
The means by which you are able to administer both voice- and nonvoice-related aspects of the system.

VROP
See "voice response output process."

W

warning
An admonishment used when there is a possibility of equipment damage.

WholeWord speech recognition
An optional feature based on whole-word technology that provides speaker independence, connected digit recognition, key word spotting, prompt interrupt, and DTMF support functionality. See "whole-word technology."

whole-word technology
The ability to recognize an entire word, not the phoneme or a part of a word. See "subword technology."

wink signal
An interruption of current to a busy lamp indicating that there is a line on hold.

word
A unique utterance understood by the recognizer.

wordlist
A set of words identified by a wordlist name. If the wordlist is part of an active vocabulary, the wordlist name appears as a recognition type in the Prompt & Collect mode field.

word spotting
The ability to search past extraneous speech during a recognition.

Index

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V