Lucent Technologies
Bell Labs Innovations

UCS 1000
R4.2

Application Development with Advanced Methods

585-313-214
Comcode 108773854
April 2000
Issue 2

Copyright and legal notices

Copyright

This document was prepared by the GLS Information Development of the EN division of Lucent Technologies. Offices are located in Denver CO, Columbus OH, Middletown NJ, and Basking Ridge NJ, USA.

Trademarks

DEFINITY, AUDIX, CONVERSANT, elemedia, SABLIME, Talkbak, Terranova, WaveLAN, MERLIN, and MERLIN LEGEND are registered trademarks and 4ESS, 5ESS, Intuity, OneMeeting, OneVision, PacketStar, PathStar, ProLogix, Lucent, Lucent Technologies, and the Lucent Technologies logo are trademarks of Lucent Technologies. Pentium is a registered trademark of Intel Corporation. Microsoft, Windows, and Windows NT are registered trademarks and Video for Windows is a trademark of

Microsoft Corporation. UNIX is a registered trademark of UNIX System Laboratories, Inc., a wholly-owned subsidiary of Novell, Inc. X Window System is a trademark and product of the Massachusetts Institute of Technology. Hewlett-Packard and HP are registered trademarks of the Hewlett-Packard corporation. Sun, Sun Microsystems, Sun Workstation, and Solaris (computer and peripherals) are registered trademarks and Solaris (operating system utilities) and Java are trademarks of Sun Microsystems, Inc. Adobe, Acrobat, Acrobat Capture, Distiller, Acrobat Exchange, Adobe Type Manager, PostScript are trademarks of Adobe Systems, Inc. Other product and brand names are trademarks of their respective owners.

Limited warranty

Lucent Technologies provides a limited warranty on this product. Refer to the “Limited Use Software License Agreement” card provided with your package.

Lucent Technologies has determined that use of this electronic data delivery system cannot cause harm to an end user's computing system and will not assume any responsibility for problems that may arise with a user's computer system while accessing the data in these document.

Every effort has been made to make sure that this document is complete and accurate at the time of release, but information is subject to change.

United States FCC compliance information

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 instructions, 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 own expense.

Part 15: Personal Computer Statement. This equipment has been certified to comply with the limits for a Class B computing device, pursuant to Subpart J of Part 15 of FCC Rules. Only peripherals (computing input/output devices, terminals, printers, etc.) certified to comply with the Class B limits may be attached to this computer. Operation with noncertified peripherals is likely to result in interference to radio and television reception.

Canadian Department of Communications interference information

This digital apparatus does not exceed the Class A limits for radio noise emissions set out in the radio interference regulations of the Canadian Department of Communications. Le Présent Appareil Nomérique n'émet pas de bruits radioélectriques dépassant les limites applicables aux appareils numériques de la class A préscrites dans le reglement sur le brouillage radioélectrique édicté par le ministére des Communications du Canada.

Toll fraud

Toll fraud is the unauthorized use of your telecommunications system by an unauthorized party, for example, persons other than your company's employees, agents, subcontractors, or persons working on your company's behalf. Note that there may be a risk of toll fraud associated with your telecommunications system and, if toll fraud occurs, it can result in

substantial additional charges for your telecommunications services. You and your system manager are responsible for the security of your system and for preventing unauthorized use. You are also responsible for reading all installation, instruction, and system administration documents provided with this product in order to fully understand the features that can introduce risk of toll fraud and the steps that can be taken to reduce that risk. Lucent Technologies 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. Lucent Technologies will not be responsible for any charges that result from such unauthorized use. If you suspect that you are being victimized by toll fraud and you need technical support or assistance, call Technical Service Center Toll Fraud Intervention Hotline at 1 800 643-2353.

Contents

5 IRAPI 184

About This Book

Overview

This book, *UCS 1000 R4.2 Application Development with Advanced Methods*, 585-313-214, is a reference for people who develop applications for the UCS 1000 R4.2 using the transaction assembler script (TAS) language, C language, and/or INTUITY Response Application Programming Interface (IRAPI). It provides information about designing software applications and writing programs that integrate the application software and system software.

Intended Audiences

The intended audiences for this book are:

- End customer application developers — This group is responsible for creating and maintaining applications in the UCS 1000 R4.2 environment.
- Custom application developers — This group is responsible for creating applications to be used in the UCS 1000 R4.2 environment for end-user customers. This audience includes any of the custom application development organizations within Lucent Technologies.

- Application distributors — This group distributes and implements applications for end-users. This audience includes independent software vendors (ISVs).

Release History

This is the first release of this book for the current release of the product.

How to Use This Book

This book is organized into the following chapters:

- Chapter 1, Application Design Considerations , provides a general understanding of the human factors as well as the hardware factors you must consider when designing an application. Chapter 1 also lists the steps involved in designing an application before you begin to process the speech data and write the script instructions.
- Chapter 2, Application Structure , provides an outline of the directory structure and naming conventions you should use when developing application programs.
- Chapter 3, TAS Script Instructions , explains the TSM process, the script conventions, the instructions used by a script, and the application-dependent functions that you can use in a script.

- Chapter 4, Data Interface Processes , explains the data interface process (DIP) interfaces between the TSM and a host or local database. This chapter describes both hard-coded and dynamic DIPs.
- Chapter 5, IRAPI , describes the INTUITY Response Application Programming Interface.
- Chapter 6, Message Logger , describes how to add or modify system messages and their associated text.
- Appendix A, Application Example, provides a complete example of the application-dependent code and the files that an application developer must develop for any speech application.
- Appendix B, Summary of TAS Script Instructions, contains manual pages for each script instruction, including the syntax, arguments, and examples.
- Appendix C, C-Library Functions, contains manual pages for each voice system C-library function, including the syntax, arguments, and examples.

Conventions Used in This Book

This section describes the conventions used in this book.

Terminology

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

  Type **y** to continue.

- The word “enter” means to type a value and then press **ENTER**. For example, an instruction to type the letter “y” and press **ENTER** is shown as

  Enter **y** to continue.

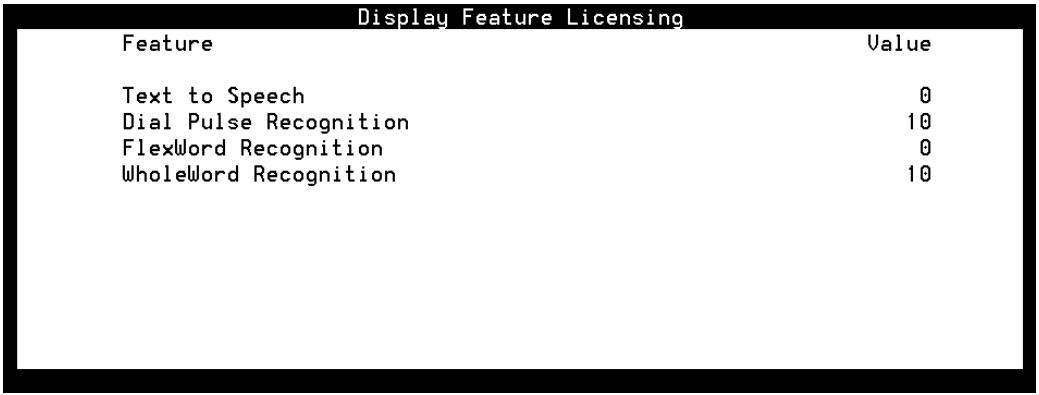
- The word “select” means to move the cursor to the desired menu item and then press **ENTER**. For example, an instruction to move the cursor to the start test option on the Network Loop-Around Test screen and then press **ENTER** is shown as

  Select **Start Test.**

- The system displays *windows, screens,* and *menus*. Windows and screens both show and request system information (Figure 1 on page xxii through Figure 4 on page xxiv). Menus (Figure 5 on page xxv) present options from which you can choose to view another menu, or a screen or window.

Example of a System Window Showing Information

Figure 1. System Window Showing Information



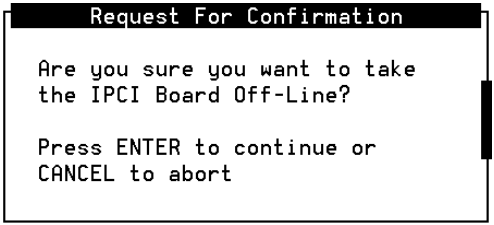
Example of a Screen Showing Information

Figure 2. Window Showing Information

```
UnixWare Installation           Primary Hard Disk Partitioning

In order to install UCS 1000 R4.2, you should reserve a UNIX system
partition (a portion of your hard disk's space) containing 100% of the
space on your primary hard disk. After you press 'ENTER' you will be
shown a screen that will allow you to create new partitions, delete
existing partitions or change the active partition of your primary hard
disk (the partition that your computer will boot from).

WARNING: All files in any partition(s) you delete will be destroyed. If
you wish to attempt to preserve any files from an existing UNIX system,
do not delete its partitions(s).

The UNIX system partition that you intend to use on the primary hard disk
must be at lease 4200 MBs and labeled "ACTIVE."

Press 'ENTER' to continue
```

Example of a Window Requesting Information

Figure 3. Window Requesting Information



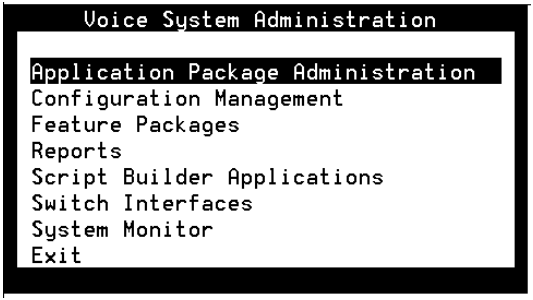
Example of a Screen Requesting Information

Figure 4. Screen Requesting Information

```
UNIX System Installation                                Set Slice
Sizes

Please select whether you would like the recommended slice
sizes or would like to customize the slice sizes.

Your choices are:
1. Recommended Slice Sizes
2. Customize Slice Sizes

Press ’1’ or ’2’ followed by ’ENTER’: 1
```

Example of a Menu Showing Information

Figure 5. A Menu



Terminal Keys

- Keys that you press on your *terminal* or *PC* are represented as rounded boxes. For example, an instruction to press the enter key is shown as

  Press **ENTER**.

- Two or three keys that you press at the same time on your *terminal* or *PC* (that is, you hold down the first key while pressing the second and/or third key) are represented as a series of separate rounded boxes. For example, an instruction to press and hold **ALT** while typing the letter “d” is shown as

  Press **ALT D**.

- Function keys on your terminal, PC, or system screens, also known as *soft keys*, are represented as round boxes followed by the function or value of that key enclosed in parentheses. For example, an instruction to press function key 3 is shown as

  Press **F3** (Choices).

- Keys that you press on your *telephone keypad* are represented as square boxes. For example, an instruction to press the first key on your telephone keypad is shown as

  Press **1** to record a message.

Screen Displays

- Values, system messages, field names, and prompts that appear on the screen are shown in typewriter-style **constant-width** type, as shown in the following examples:

  Example 1:

  Enter the number of ports to be dedicated to outbound traffic in the **Maximum Simultaneous Ports** field.

  Example 2:

  Alarm Form Update was successful.

  Press <Enter> to continue.

- The sequence of menu options that you must select to display a specific screen or submenu is shown as follows:

  Start at the Main Menu and select

  In this example, you would access the Main Menu and select the Voice System Administration menu. From the Voice System Administration menu, you would then select the Configuration Management screen.

- Screens shown in this book are examples only. The screens you see on your machine will be similar, but not exactly the same.

Some Screen Simulations

Text in a simulated screen display appears in `type-writer text`.

Example:

```
            QuickStart - Data Recovery Rescue
Copyright(c) 1997-1999 by Enhanced Software Technologies, Inc.
      Serial# 8200-999                          Version: 1.3.17

   ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐
   │Backup    │ │Verify    │ │Recover   │ │Duplicate │ │Configure │ │Exit      │
   │System    │ │System    │ │System    │ │Diskette  │ │QuickStart│ │and Reboot│
   └──────────┘ └──────────┘ └──────────┘ └──────────┘ └──────────┘ └──────────┘
```

Items That May or May Not Appear

Grayed-out type represents optional items that may or may not appear in a given display.

Example:

Once the backup is complete, the system displays a message similar to the following:

`The Differential UNIX backup is now complete. Please remove the tape and label it as "Differential UNIX Backup, created April 30, 1999."`

Cross References and Hypertext

Blue, underlined type indicates a cross reference or hypertext link that will take you to another location in the document when you click on it.

Other Typography

- Commands and text you type in or enter appear in **bold type**, as in the following examples:

  Example 1:

  Enter **change-switch-time-zone** at the **enter command:** prompt.

  Example 2:

  Type **high** or **low** in the **Speed:** field.

- Command variables are shown in ***bold italic*** type when they are part of what you must type in and *regular italic* type when they are not, for example

  Enter **ch ma *machine_name***, where *machine_name* is the name of the call delivery machine you just created.

Safety and Security Alert Labels

This book uses the following symbols to call your attention to potential problems that could cause personal injury, damage to equipment, loss of data, service interruptions, or breaches of toll fraud security:

CAUTION:
Indicates the presence of a hazard that if not avoided *can* or *will* cause minor personal injury or property damage, including loss of data.

WARNING:
Indicates the presence of a hazard that if not avoided *can* cause death or severe personal injury.

DANGER:
Indicates the presence of a hazard that if not avoided *will* cause death or severe personal injury.

SECURITY ALERT:
Indicates the presence of a toll fraud security hazard. Toll fraud is the unauthorized use of a telecommunications system by an unauthorized party.

Related Resources

This section describes additional documentation and training available for you to learn more about the product.

Documentation

The *UCS 1000 R4.2 System Description*, 585-313-209, contains a detailed description of all books included in UCS 1000 R4.2 documentation library. Always refer to the appropriate book for specific information on planning, installing, administering, or maintaining a UCS 1000 R4.2.

Updates to the Product

The following Web site displays any updates or exceptions to the product that have occurred after the publication of this document:
http://glsdocs.lucent.com

Recommended References

To completely develop and maintain an application script, the following books in the documentation library will be useful.

- *UCS 1000 R4.2 Administration*, 585-313-507
- *UCS 1000 R4.2 Speech Development, Processing, and Recognition*, 585-313-212
- *UCS 1000 R4.2 Communication Development*, 585-313-213

Training

For information on UCS 1000 R4.2 training, check the Lucent Message Institute website at: *http://www.octel.com/octelu/index.html*

Using the CD–ROM Documentation

Lucent Technologies ships the documentation in electronic form. Using the Adobe® Acrobat® Reader application, you can read these documents on a Windows PC, on a Sun Solaris workstation, or on an HP-UX workstation. Acrobat Reader displays high-quality, print-like graphics on both UNIX and Windows platforms. It provides scrolling, zoom, and extensive search capabilities, along with online help. A copy of Acrobat Reader is included with the documents.

Note: If viewing documents online, it is recommended that you use a separate platform and not the UCS 1000 R4.2.

Setting the Default Magnification

You can set your default magnification by selecting **File | Preferences | General**. We recommend the **Fit Page** option.

Adjusting the Window Size

On HP and Sun workstations, you can control the size of the reader window by using the **-geometry** argument. For example, the command string **acroread -geometry 900x900 mainmenu.pdf** opens the main menu with a window size of 900 pixels square.

Hiding and Displaying Bookmarks

By default, the document appears with bookmarks displayed on the left side of the screen. The bookmarks serve as a hypertext table of contents for the chapter you are viewing. You can control the appearance of bookmarks by selecting **View | Page Only** or **View | Bookmarks and Page.**

Using the Button Bar

The button bar can take you to the book's Index, table of contents, main menu, and glossary. It also lets you update your documents. Click the corresponding button to jump to the section you want to read.

Using Hypertext Links

Hypertext-linked text appears in blue, italics, and underlined. These links are shortcuts to other sections or books.

Navigating with Double Arrow Keys

The double right and double left arrows (◄◄ and ►►) at the top of the Acrobat Reader window are the go-back and go-forward functions. The go-back button takes you to the last page you visited prior to the current page. Typically, you use ◄◄ to jump back to the main text from a cross reference or illustration.

Searching for Topics

Acrobat has a sophisticated search capability. From the main menu, select **Tools | Search.** Then choose the **Master Index.**

Displaying Figures

If lines in figures appear broken or absent, increase the magnification. You might also want to print a paper copy of the figure for better resolution.

Printing the Documentation

If you would like to read the documentation in paper form rather than on a computer monitor, you can print all or portions of the online screens.

You can also order the printed documents by calling 1-888-582-3688 or visiting the Customer Information Center (CIC) website at: *http://www.lucentdocs.com/cgi-bin/CIC_store.cgi*

Printing an Entire Document

To print an entire document, do the following:

1. From the documentation main menu screen, select one of the print-optimized documents. Print-optimized documents print two-screens to a side, both sides of the sheet on 8.5x11-in or A4 paper.
2. Select **File | Print**.
3. Enter the page range you want to print, or select **All**. Note that the print page range is different from the page numbers on the documents (they print two to a page).
4. Close the file. Do not leave this file open while viewing the electronic documents.

Printing Part of a Document

To print a single page or a short section, you can print directly from the online version of the document.

1. Select **File | Print**.
2. Enter the page range you want to print, or select **Current**.

   The document prints, one screen per side, two sides per sheet.

How to Comment on This Book

We are interested in your suggestions for improving this book. Please complete and return the reader comment card located at the end of the book.

If the reader comment card has been removed, send your comments to:

Lucent Technologies
GLS Information Development Division
Room 22-2H15
11900 North Pecos Street
Denver, Colorado 80234

You may also fax you comments to the attention of the Lucent Technologies UCS 1000 R4.2 writing team at (303) 538-1741.

Please mention the name and order number of this book, *UCS 1000 R4.2 Application Development with Advanced Methods*, 585-313-214.

1 Application Design Considerations

Overview

This chapter describes general points to consider when developing an application using the IRAPI library functions, the transaction assembler script (**tas**) language, and/or C-language. Specific procedures for developing application programs are covered later in Chapters 3 through 6.

Designing a Successful Application

A successful application meets the following criteria:

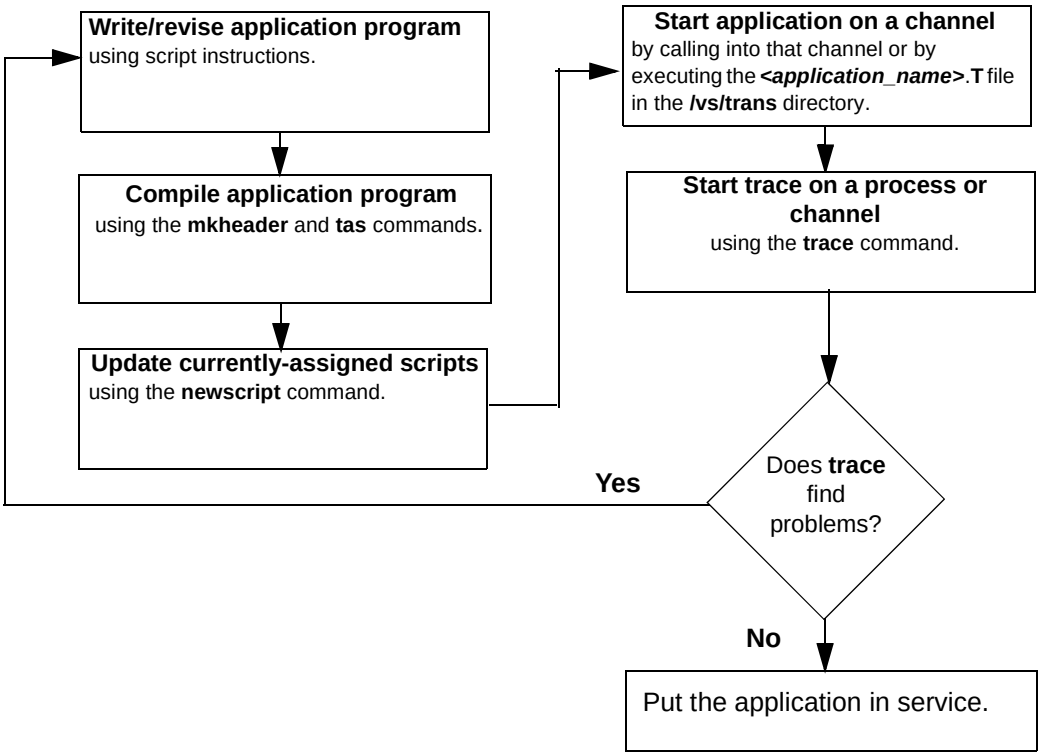
- The end user can easily access and use the service offered.
- The end user is provided with the necessary information.

To design an application to meet these criteria, you must consider the overall system including the UCS 1000 R4.2, the end user, and the data source (may be a separate host machine).

Application Development Tools

Figure 6 on page 3 illustrates the typical steps in developing an application and specifies the tools to use at each step.

Figure 6. Using Application Development Tools — Example



This book describes applications created using an editor to write scripts in the **tas** instruction language and compiling them with the **tas** command and by using IRAPI, and C-language code.

The standard set of tools available to the application developer include the UnixWare operating system, speech administration tools, **tas** commands, and debugging tools.

UnixWare Operating System Tools

The UnixWare tools include the **vi** editor, and standard UNIX commands such as **grep**, **cat**, etc.

See the UnixWare documentation set for more information about standard UNIX tools.

File Processing Program Tools

Several commands/programs are designed to help you process files for application development. These include:

- **mkheader** — This command creates files for the application to define memory used by the TAS script (see Chapter 3, TAS Script Instructions).
- **tas** — This program accepts a file in script source code and produces a TSM executable file (see Chapter 3, TAS Script Instructions).
- **newscript** — This command processes changes to all currently assigned scripts. If you write an application using script language and use **tas** to assemble the script, you must use **newscript** to ensure that the most recent version of the script is used.

Speech Administration Tools

Commands such as **add**, **copy**, **erase**, and **list** are among those tools available to develop and edit speech files located in the default filesystem, **/home2/vfs/talkfiles**.

Debugging Tools

Debugging tools include **trace**, **truss**, **debug**, and **untas**. The **trace** script instruction monitors specific DIPs and/or channels while the script is running and stores the information in a buffer; use the **trace** command to display this information.

For more information about the **trace** script instruction, see Appendix B, Summary of TAS Script Instructions. See Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration,* 585-313-507, for more information about the **trace** command.

The **irTrace**(3IRAPI) functions support a variety of tracing operations including:

- Channel level tracing
- Process level
- All errors tracing
- Compatibility with **db_put** (**tas** debugging tool) tracing for output to the **trace** command

2 Application Structure

Overview

This chapter provides application developers with the information required to name, organize, and structure files when developing an application. It contains an overview of the basic tools available with the system, and includes information to help you:

- Develop speech files and application programs
- Organize files
- Establish naming conventions for files
- Develop a coding style that is easy to maintain

Application Components

Each application used with the UCS 1000 R4.2 consists of one or more of the following program components:

- Script

  A script functions as a set of instructions the transaction state machine (TSM) process uses to run the application.

- Data interface process (DIP)

  A DIP performs operations not easily performed in script instructions, such as extensive calculations or interfacing to a host machine. You must write a DIP module in C-language. Writing a DIP also requires an understanding of the UnixWare operating system. See Chapter 4, Data Interface Processes, for more information on writing a DIP.

- IRAPI library functions

  These functions provide high-level, C-language interfaces to accomplish both voice-processing and telephony functions.

- **extend** functions

  The extend function is a TSM instruction that executes customer-written, C-language code.

- speech files

Conventions for Naming Files and Programs

To make files easy to identify and to meet the requirements of the application compiler, the system uses naming conventions for the files and programs. Most of the naming conventions consist of prefixes and suffixes that make the programs and files easy to classify into a group or type. The application name is often part of the name of the file or the program.

Table 1 describes files and program names; information provided by the application developer is shown in ***bold-italics***.

Table 1. File and Program Naming Conventions

File and/or Program	Description	Examples
***name*.c**	This is a C-language source program.	**hostmeas.c** or **msg1hlr.c**
***name*.o**	This is a compiled C-program in which external references are not resolved.	**hostmeas.o** or **msg1hdlr.o**
		1 of 4

Table 1. File and Program Naming Conventions

File and/or Program	Description	Examples
***name*.h**	This is a header file that contains structures and identifier definitions that do not require space allocation. This allows separately developed modules to use the same header files without repeating header file references in several places.	**hwrtype.h**
DIP*N*	Each DIP is referenced by the name **DIP*N*** where *N* is a number or a word. See Chapter 4, Data Interface Processes, for more information on DIPs.	**DIP0** or **DIP_test**
***application_name*.t**	This is a script source file for *application_name*.	**stock.t**
		2 of 4

Table 1. File and Program Naming Conventions

File and/or Program	Description	Examples
***application_name*.T**	This is an executable TAS script that has been processed using the **tas** command with ***application_name*.t** as an argument, or an executable IRAPI application transaction definition file created with the **defService** command.	**stock.T**
***application_name*def.h**	This header file defines the application-dependent user memory for the TSM. The file is produced by running the associated executable version of ***application_name*_aloc.c** or by using the **mkheader** command.	**stock_aloc.c**
		3 of 4

Table 1. File and Program Naming Conventions

File and/or Program	Description	Examples
***application_name*_aloc.c**	This application-dependent program allocates user memory for the database structures in the script. The script uses the structures as temporary work spaces and for communicating with the internal data processes. When the program is executed, it produces the header file ***application_name*def.h**. This header file defines the addresses of variables used by TSM. The **mkheader** command is used in creating and executing this program.	**stock_aloc.c**
***application_name*.D**	This file contains descriptions of application variables that normally are used as event counters.	**stock.D**
		4 of 4

Coding Style

Establishing a consistent coding style makes the programs and scripts readable by other developers and makes debugging and maintaining them easier and quicker. Recommendations are made here concerning define statements, enum definitions, labels, and inline comments.

Define Statements

Define statements used in naming addresses and numerical data make the program more understandable by explaining a value. For example, referring to the value -10 as MISTAKE is easier to interpret and understand:

```
#define MISTAKE ( -10)/* 1 of 15 values returned by getname */
.
.
.
.
jmp(r.3==MISTAKE, CORRECT)
```

You can put define statements in the header files and in the program by using C-language **#include** statements that link the definitions to the program code during assembly or compilation. Use a define statement only once for the same memory location or value. By convention, define statements are in uppercase letters. They may have underscores (_), but no embedded spaces.

You can use one file with a set of defines for both the script and a DIP. This ensures consistency within an application and makes it easier to change the defines.

Note: If your script contains a large number of define statements, TAS may report messages such as the following during compilation, where *script.t* is the script source file and *1068* is the line in which the define appears:

```
script.t: 1068: too much defining
```

The limit to the number of define statements that a script can have depends on the number of defined macros and their size. If this type of message appears, reduce the number of define statements in your script.

The C preprocessor symbol __TAS__ is defined for TAS scripts. It may be used in source files used by both TAS and the C compiler.

Enum Definitions

The **tas** compiler supports C-language enum constant definitions commonly used in header files. Therefore, you can use enum constant names whenever you use a **#define** constant.

Script Labels

A label is a C-style identifier followed by a colon (:). It marks the instructions that follow it. By convention, labels for major blocks of code are in uppercase letters. Labels for subordinate blocks of code are in lowercase letters. All labels must begin with an alphabetic character.

Some examples of labels are

```
GREET:
talk(“hello”)
rts( )
```

```
GET_ID:
/* COMMENTS */
jmp( r.3 == 0, strt_idloop )
...
```

```
strt_idloop:
getinput( ch.DG, 9)
...
rts( )
```

The uppercase labels GREET and GET_ID identify major blocks of code or subroutines. The lowercase label, strt_idloop, identifies a block of code under the main subroutine GET_ID.

Inline Comments

Inline comments should either precede or be to the right of those lines of code where an explanation would be useful. For example, an appropriate comment for a **goto** script instruction or a subroutine call might be “cleanup routine” or “send voice response” to reflect the destination. Or, using the example given above for script labels, the comments for the GET_ID subroutine might be:

```
GET_ID:
/* This subroutine collects digits from the caller */
jmp( r.3 == 0, strt_idloop )
...
```

3 TAS Script Instructions

Overview

This chapter describes the conventions used in writing a script, along with all the transaction assembler script (**tas**) instructions needed to develop the application script. It provides application developers with the information required to use the **tas** language for developing an application.

Transaction State Machine

The transaction state machine (TSM) software process is an IRAPI application that manages the execution of **tas** language applications.

A **tas** language application is comprised of a set of script instructions and commands. These script instructions, running within the TSM software, are a sequence of library function calls that manage the low level interactions required to operate the system. At any one time, the system may run several occurrences of the same script as well as the execution of several scripts concurrently within the TSM process.

Based on the arguments in the script instructions, TSM uses IRAPI function calls to send messages to the system devices and other software processes that control the access to system hardware or a local or host database.

A TSM script begins to execute when a call is recognized by the Application Dispatch (AD) process on a channel to which a TSM application is assigned. TSM gains control of the channel for the script and processes script functions through the IRAPI. TSM returns ownership of the channel to the AD process when the call ends.

Both the script and TSM collect call information while a call is in progress. At the end of a call, TSM combines its data with the script data and sends the information to the call data handler (CDH). The CDH makes the information available in reports to the host and the UCS 1000 R4.2.

A script can be assembled (using **tas**), loaded, changed, or replaced without affecting the other scripts running on TSM or other IRAPI applications running on the system. To insure that TSM loads the revised script, the **newscript** command should be used (see Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration*, 585-313-507, for more information about **newscript**).

The Script and Call Progression

This section describes how the system processes a call using a TSM script.

1. The system accepts a call from the central office or switch on an E1/T1 circuit card.
2. TheE1/T1 circuit card then signals the T1 interface process (TWIP), that it has accepted the call.
3. TWIP signals the AD process it has accepted the call.
4. AD checks a service table to determine the service and process required for the call. In this example, TSM is the selected process.
5. TSM reads the script instructions and allows the script to control the sequence of events during a call.
6. AD takes control when the call has ended.

The following sections describe call progression with a script in greater detail.

Call Progression Starting Conditions

Before the script takes control, the following sequence of conditions must be met:

- A caller dials and reaches the system's incoming telephone facility.

- The software recognizes the ringing condition.
- AD checks an internal table to determine the script to run for the call.
- All script memory is set to zero and the time-outs are set to their default values.

Script Control

When all the starting conditions are met, the script takes control and typically executes the following functions during the call:

- Answers the incoming line (takes it off hook)
- Sends recorded voice messages to the caller
- Listens to touch-tone signals
- Accesses information from the host or from the disk
- Sends information to a host or a local database
- Records transaction events on disk
- Takes action when a caller does not respond
- Signals a termination condition to TSM

TSM Control

TSM terminates an active script if one of the following conditions exists:

- The script executes a **quit** instruction.
- The maintenance software provides commands to seize control of equipment at any time, thereby terminating transactions on one or more channels without notice.
- A program error causes a script to terminate, such as when a **goto** or **quit** is missing after the last instruction.

Script Termination

When the script ends, TSM performs the following functions:

- Puts the telephone line on hook if the script did not disconnect
- Stops voice play
- Discards any pending messages from the host
- Sends the CDH a message about the transaction and a copy of the event memory
- Releases the channel to AD so it will be available for the next call

Text-to-Speech and the LSPS II

Be aware of the following limitations regarding TAS instructions and the LSPS ii circuit card.

String Length Limit

When Text-to-Speech is on the LSPS II circuit card, the maximum length for a string to be played is 128 characters. Any additional characters are truncated. To play strings longer than 128 characters, do one of the following:

- Save the strings in a text file and then use Text-to-Speech functions, such as fsay or irFSay(3IRAPI) to play them.
- Split the strings into several strings that are 128 characters or fewer.

TSM Instructions Limit

By default, TSM applications can queue no more than 256 say, fsay, talk, ftalk, tchars, or tnum TAS instructions. If this limit is exceeded, TSM will force queued speech requests to be played. If the LSPS II circuit card is used for play or Text-to-Speech, then you must decrease the default limit of 256 to 64, or unexpected results will occur. To do this, add the following line to the **/vs/data/irAPI.rc** file:

SPEECH_QUEUE_LIMIT=64

Data Storage

The script has the following four areas where it temporarily stores data for each call it handles. TSM clears these areas at the beginning of a new call.

- User memory

  User memory is a work area for the script to store database information, global variables, and data sent to and from the host.

  The script writer is responsible for partitioning user space. This must be done carefully by assigning data addresses or by using the tool **mkheader**, discussed in Chapter 2, Application Structure , and *UCS 1000 R4.2 Administration,* 585-313-507.

  Each script is allocated 512 bytes for user space, but automatic allocations ensure up to 51,200 bytes if script data defines require additional space.

- Event memory

  The event memory contains a record of the events that occurred for each transaction. Event memory consists of 100 32-bit integers.

- Registers

  Sixteen registers, *r.0* through *r.15*, allow the script to manipulate data outside of user memory. Three of the registers perform special functions.

Register *r.0* (and occasionally *r.1* and *r.2*) is a return register that can be used to indicate the results of a specific instruction. For example, the **dbase** instruction (described under Script Instructions on page 34) sets *r.0* to a positive number on successful completion, which indicates the message contents. In general, a negative number indicates that the instruction failed. For example, if a database instruction that is supposed to receive data did not return any data, then *r.0* is set to -2 after an instruction time-out period (45 seconds by default). See the **nwitime** instruction later in this chapter or in Appendix B, Summary of TAS Script Instructions.

Registers can also be used for indirect addressing.

Note: Because most of the instructions store return values in *r.0*, it is recommended that this register not be used for general purposes.

Register *r.2* and *r.3* are used to pass information to subroutines when a subroutine call is made with up to two arguments specified. The called subroutine reads the first field of information from *r.3* and the second field from *r.2*.

- Stacks

  A stack is a set of data storage locations that are accessed in a fixed sequence. The contents of r.1 through r.15 are saved on a stack when a subroutine is called. Upon return from the subroutine, they are reloaded with the stack values.

Call Data Collection

Both the script and TSM collect call data during a transaction. The script can store application data in event memory and save any application-related data. The data might be response time, user ID, request types, number of invalid selections, and an event counter. TSM collects generic data such as the script name, channel number, start time, and stop time and stores it in a call data record.

At the end of a call, TSM copies the generic data it has collected and the contents of event memory into a call data record and sends it to call data handler (CDH). Call data is stored in the ORACLE database.

The reports generated from the database are available to the system operator (see “Reports” in Chapter 8, “Daily Administration”, *UCS 1000 R4.2 Administration*, 585-313-507).

The .D File

The **.D** file provides descriptive labels for events when the events are displayed. The event counter array space may contain event counter integers, strings, or both. Records beginning with an integer between 0–99 are interpreted as valid event specification records. You do not have to use a 0 or 1 as the first event counter.

The following is an example of the **.D** file syntax, where *WS* refers to a tab or blank space and *STR* refers to the literal string STR:

<*event_number*> [<*WS*> *STR*] [<*WS*> <*label_string*>]

The following is an example of the **.D** file syntax, where event memory 1 stores string data and displays it under the label “User Name”:

1 STR User Name

A sufficient amount of event memory space for storage of the strings should be allocated. This includes 1 byte for the null character at the end. In addition, the contents of one event counter should not overlap the contents of another event counter. You should also make sure the script copies the string starting at the event counter specified in the **.D** file.

Event data is reported only if it is specified in the service script and the file **/vs/trans/<*script_name*>.D** exists in the proper format. Place all **.D** files in the **/vs/trans** directory. It is important to place strings in the call event space properly. You must know the length of the string and map it correctly onto the 4-bit events of the event space. Use the command **/vs/bin/cddrpt** to view script events.

Script Conventions

This section provides rules that must be followed when writing scripts. They include the syntax, argument structure and types, and address mode and format for script instructions.

Script Syntax

The syntax used for a script instruction is an instruction mnemonic followed by any required and optional arguments enclosed in parentheses. There are some conventions that appear only in this book and are not part of an actual script. Brackets ([ ]) indicate an optional argument for an instruction and the less-than or greater-than symbols (< >) indicate a label instruction for a program segment.

Table 2 lists the characters used in the script syntax.

Table 2. Script Syntax Characters

Character	Meaning	Example
parentheses ( )	Encloses arguments in an instruction	**load (ch.ONE,ch.TWO)**
comma ,	Optional character that separates the arguments of an instruction (you may also use spaces to separate arguments, but the use of commas is strongly encouraged)	**tchars (ch.ONE, 'F')**

1 of 2

Table 2. Script Syntax Characters

period .	Required syntax character that separates an argument type and argument value	r.1 — argument type is a register, register number is 1
asterick * or ampersand &	Preceding a type, signifies that the value is the number of a script register containing the user space address; type without an asterisk or ampersand signifies that the value is an address	***ch.1** — character string at address in register 1 **ch.1024** — character string at address 1024

2 of 2

Destination and Source Arguments

The address modes are represented in Table 3.

Table 3. Destination and Source Addresses

Address	Modes
type.dst	All types except immediate and time
type.src	All types

1 of 2

Table 3. Destination and Source Addresses

Address	Modes
ctype.dst	Only types char and *char
ctype.src	Only types char and *char

2 of 2

Arguments to Script Instructions

Acceptable abbreviations for argument types are shown in the Table 4 on page 30. The following is an example of an argument format, where type is one of the argument types listed in Table 4 on page 30, and # is a numerical value or a define statement or an enum symbol:

type.#

You may write numerical values in decimal (256) or hexadecimal (0x100) notation. See Define Statements on page 12 in Chapter 2, Application Structure, for additional information about define statements.

Address Modes

The data types are summarized in Table 4 on page 30. The values associated with character, short, and integer types represent user space addresses defined in the script or in header files.

Most of the script instructions do not check data typing. Thus, in most instances, the outcome of using character, short, or integer typing has no effect on the outcome of the instruction. The instructions are sensitive, however, to the contents of the specified user space locations. If characters are required, a null-terminated ASCII string must start at the specified address. Similarly, a short integer (2 bytes) or long integer (4 bytes) must start at the specified address if the instruction requires an integer value.

The subsequent instruction descriptions indicate when character values result in or are required by the *ctype.dest* or *ctype.src* descriptions. The *type.dest* or *type.src* descriptions require short or long integer values. Only the **atoi** and **itoa** instructions convert characters to integers and integers to characters.

Although most instructions allow character typing for integer values and integer typing for character values, this practice should be avoided. Integer types must be assigned to even user space byte addresses while character strings may begin at even or odd locations. The integer types are assigned values ranging from -32768 to 32767 (short) and -2147483648 to 2147483647 (int).

In general, an integer variable type (int or short) may be used anywhere that an integer constant (immed) is used. Integer variables allow more flexibility with instruction arguments, but TSM requires more time to retrieve the integer variable values.

Table 4. Argument Data Types

Argument Type	Field Width (bytes)	Meaning	Example
immed*	—	Actual value, for example, a number, string, or string address	4, ”xyz”, 64,”ABC”
time	4	Operating system time value. A value following (.) is ignored.	t.0
reg	4	Contents of script register	r.1
char	1	Character address in user memory	ch.VARIABLE
short	2	Short address in user memory	sh.SHORT
int	4	Integer address in user memory	int.NUMBER
event	4	Address in event memory[2]	ev.1

1 of 4

Table 4. Argument Data Types

Argument Type	Field Width (bytes)	Meaning	Example
*char	1	Register containing address of a character in user memory	*ch.1
*short	2	Register containing address of a short in user memory	*sh.1
*int	4	Register containing address of an integer in user memory	*int.1
*event	4	Register containing address in event memory†	*ev.1
&char	1	Register containing address of a character in parent script user memory‡	&ch.1
&short	2	Register containing address of a short in parent script user memory3	&sh.1

2 of 4

Table 4. Argument Data Types

Argument Type	Field Width (bytes)	Meaning	Example
&int	4	Register containing address of an integer in parent script user memory[3]	&int.1
script	-	Name of script (string). A value following (.) is ignored.[3]	script.o
&script	-	Name of nearest parent script (string). Null string if there is no parent script. A value following (.) is ignored.[3]	&script.o

3 of 4

Table 4. Argument Data Types

Argument Type	Field Width (bytes)	Meaning	Example
X	-	Data passed via the exec instruction. A value following (.) is ignored. See **exec** in Appendix B, Summary of TAS Script Instructions.	X.0
Chan	4	The channel number on which the script is running. A value following (.) is ignored.	Ch.0

4 of 4

* The immed argument type specification is optional. Arguments with no type specification are assumed to be immed.

† Event memory is write-only for the script, since the buffer for this data is maintained inside the IRAPI library, not TSM.

‡ See details for the subprog instruction in Appendix B, Summary of TAS Script Instructions

Numbers following the dot (.) in argument specification may also be expressed as simple arithmetic expressions involving addition and subtraction of integer constants. For example, the argument

char.VARIABLE+12 refers to the character string starting at the user memory offset 12 bytes after the offset defined by the VARIABLE symbol. You can also use C-preprocessor # define symbols, positive and negative numbers, and parenthetical expressions.

Script Instructions

The following sections detail the script instructions according to similarities in functionality.

Voice Output Instructions

In this section, instructions that control speech output are described. These instructions send voice data to the E1/T1 and speech and signal processing (SSP) or LSPS II circuit cards. Each description is followed by a brief example using that instruction. An example at the end of this section illustrates how the instructions described here might be used in a script.

- **tfile("*listfile 1*"[,"*listfile2*" ...])**

  The **tfile** instruction specifies the speech database to use for the script. The first phrase listfile name, called *listfile 1* (see Chapter 2, Application Structure), is the name of the primary listfile. Its talkfile number is the default talkfile for speech referenced by phrases and is used for **tnum**,

tchar, and **talk** instructions if the talkfile portion of the phrase ID is 0 (unless the talkfile number is changed later by a **setalk** instruction).

Each phrase in the talkfile is identified by a unique number and string in the phrase listfile. Because TAS uses this information, the **tfile** instruction must be specified in the script before the first voice output instruction.

The phrase listfile usually is named ***application_name*.pl**. Phrases in the primary listfile are not bound to the talkfile when the script is compiled. They will be played from the talkfile currently in effect when the **talk** instruction is executed. However, any additional listfiles given in the file instruction have the talkfile and phrase number bound when the script is compiled. Phrases selected from these listfiles are not affected by changes in the talkfile that occur during script execution.

The following is an example of the **tfile** instruction:

```
tfile("list.stocks")
```

This instruction tells the TAS to use the list.stocks speech database for the next transaction.

- **tchars(*ctype.src*[,*type.inflection*])**

  The **tchars** instruction puts its first argument into a queue for speaking. The first argument is a null terminated string of alphanumeric characters. This character string is spoken character-by-character, for example, letters and digits. The second argument, when specified, controls the speech inflection.

  Inflection is indicated in Table 5. The default for the inflection is m for medial.

Table 5. Speech Inflection Values

Inflection	One Phrase	Multiple Phrases
r	Rising	Rising on first; medial on others
m	Medial	Medial on all
f	Falling	Falling on last; medial on others
t	Falling	Rising on first; falling on last; medial on others

The following example of the **tchars** instruction directs the script to speak the contents of INITIALS with falling inflection on the last character and medial inflection on all other characters.

```
tchars(INITIALS,'f')
```

- **tnum(*type.src{[,type.inflection][,type.style]}*)**

  The **tnum** instruction puts the phrases that speak the numeric value, specified by the first argument, in a queue. It interprets the numeric value of the first argument and selects recorded phrases that say the number in a natural way. For example, 202 is a number that is spoken as a single phrase — two-hundred-two. The second argument, when specified, controls the speech inflection. The optional *type.style* argument specifies the speech coding style.

Note: The **tnum** instruction does not interpret numeric values in any language other than English because the rules for concatenating numbers varies depending on the language. The Enhanced Basic Speech package currently includes numbers 1–20, 30, 40, 50, 60, 70, 80, 90, 100, 1000, and 10000. This method forms numbers by combining these standard phrases.

The **tnum** instruction uses the same arguments for inflection as the **tchars** instruction (see Table 5 on page 36).

The **tnum** instruction does not support speaking numbers in the billions and trillions because most of these numbers are too big to fit into an integer variable. However, the phrases “billion” and “trillion” are included in the Enhanced Basic Speech package. If your script requires such large numbers, we suggest that you start with an ASCII string, parse the string (getting the amounts of billions and trillions as substrings), then convert the three resulting substrings to integer values and speak them with the **tnum** instruction. Insert a **talk** instruction with the phrase for “trillion” or “billion,” where appropriate.

In the following example, the **tnum** instruction tells the script to speak the numeric value of int.FOUR with falling inflection on the last character and medial inflection on all other characters:

```
load(int.FOUR, 4)
tnum(int.FOUR,'f')
```

- **talk(type.src*[,type.style]*)**

  The **talk** instruction uses the *type.src* argument to specify the phrase to be spoken. The second optional argument, *type.style*, is the coding style of the speech to play back. For the LSPS II circuit card, all speech to be played in one group of requests must use the same coding style.

Two examples of the **talk** instruction are:

```
talk(10)
talk(11)
```

"Hello this is the Lake Server"
"Please enter your ID "

To simplify writing the **talk** instructions used in matching the phrases in the ***application_name*.pl** file with the *"phrase_name"* arguments in the **talk** instruction, the **talk** arguments may be abbreviated. In this process, except for the final period or question mark, punctuation is for reference only and is ignored. Each character or word must be separated by a space. Also, uppercase characters are converted to lowercase characters.

Two ways of writing the **talk** instruction for the first example are:

```
talk("Hello, Lake Server")
talk("h, l s")
```

Words may be eliminated, but the words or abbreviations used must be written in the same order as in the ***application_name*.pl** file. They will match as long as the argument has enough of the key words in the desired phrase.

The following examples illustrate an abbreviated **talk** instruction.

```
talk("h l s")
talk("Hello Lake Server")
```

Only the first letter of a word needs to be used in matching a phrase. Note in the following examples that although each phrase would match, a person reading these instructions would find it helpful to see more than just the first character.

```
talk("H I 1")
talk("H I 1 S")
```

Although only the first letter of each word must be specified, it is recommended that you spell the phrase to the extent that it is uniquely identifiable.

- **talk(*type.src*)**

  This version of the **talk** instruction can be used where there is a variable phrase number. Instead of entering the *"phrase_name"* to identify the speech to be queued for the Tip/Ring and the SSP circuit card, the corresponding number found by the *"phrase_name"* in the ***application_name*.pl** file can be used.

An example of the **talk** instruction is:

```
#define PHRASE 40
.
.
.
talk(int.PHRASE) /*speaks the application_name.pl file*/
/*phrase the number of which is found at*/
/*address 40*/
```

- **tflush([*must_hear_flag][,wait_indicator][,remember_flag]*)**

  The **tflush** instruction typically follows a **talk, ftalk**, **tnum**, or **tchars** instruction to force queued phrases to be spoken that could otherwise be terminated by a touch-tone signal sent by the caller. Under normal operating conditions, a touch-tone signal terminates any speech activity (voice play or voice coding) on that channel. (This feature usually is referred to as talkoff.) Integer variables or registers, as well as literals, may be used for arguments to **tflush**.

  The **tflush** instruction also causes queued speech to be output as do the other wait causing instructions. Thus, **tflush** can be used to force speech to be spoken at appropriate points in the script.

  The three optional arguments to **tflush** can be set to the values listed in Table 6 on page 42. If **tflush** is used without any arguments, the default value of 0 is used for all arguments.

Table 6. tflush Arguments

Argument	Value	Value Result
must_hear_flag	0	Touch tones entered during play or voice coding cause play or voice coding to stop (default).
	1	Touch tones entered during play or voice coding do not cause play or voice coding to stop.
wait_indicator	0	Wait for the play to complete before continuing script execution (default).
	1	Do not wait for the play to complete. Continue script execution immediately after queuing.
*remember_flag**	1	Remember phrases played by this instruction so they may be played again with the **talkresume** instruction.
	0	Do not remember the speech.

* The remember flag has no effect when playing TTS.

The *must_hear_flag* option, when set to a non-zero value, disables talkoff so that speech activity (voice play or voice coding) on the current channel is not stopped by touch tones. When this option is used with speech play-related instructions (**talk**, **tnum**, **tchars**), a **tflush(1)** should follow those instructions. When using **tflush** with voice coding (**vc**), **tflush(1)** should precede the **vc** instruction. The talkoff is enabled automatically by the next wait-causing instruction in the script (see Flow Control Instructions on page 70 for a list of wait-causing instructions).

Note that if talkoff is disabled, speech play may interfere with incoming touch tones. Unless the **setttfl** instruction is used to enable the type-ahead feature, playing new speech causes any touch tones that have been typed up to that point to be deleted.

The **tflush** *wait_indicator* option, when set to a non-zero value, allows the script to start a play, then continues script execution immediately without waiting for completion of the play. By using a *wait_indicator* of 0, which is the default, the script does not start execution until a play complete message is received.

The **tflush** instruction stores a return value in register 0. If the value is negative, an error has occurred. If the value is +1, the play complete was caused by talkoff. If the value is 0, play completed successfully.

The following are examples of the **tflush** instruction:

```
talk("You must hear this announcement before continuing")
tflush(1) /*does not end play if caller enters a */
    /*touch tone*/

tflush(1) /*do not end coding if user enters touch */
    /*tones*/
vc('b',10,ADPCM32)
```

Note: In the second example, any touch tones entered are encoded along with the speech.

- **talkresume(*type.offset*)**

  The **talkresume** instruction plays whatever phrases remain from the last **tflush** instruction starting at the point they were interrupted (that is, by talkoff) plus the given offset in seconds. If the offset is a positive number, speech is played from a point after the interruption. If the offset is a negative number, speech is played from a point before the interruption. If the offset is 0, play starts at the point where the interruption occurred. If all of the phrases have been played, only a negative offset has any effect. For example, this allows a developer to include a *fast forward* or *rewind* feature into speech playing.

The **talkresume** instruction stores a return value in register 0. If the value is negative, an error has occurred. If the value is 0, play completed successfully. If the value is +1, the *play complete* was caused by talkoff. If the value is +2, there was no speech left to play (that is, **talkresume** was given with a non-negative offset when all the speech had been played already).

For **talkresume** to work properly, the speech it affects must have been played originally with the **tflush** instruction with the optional *remember_flag* argument set to 1. This tells the system to remember the speech that **tflush** tells it to play and to keep track of where that speech is interrupted. Subsequent calls to **talkresume** then have the desired effect on this speech. The system remembers the speech it was playing until it receives another set of phrases to play by subsequent script instructions. Only one set of phrases can be remembered per channel at a time. (Here, a set of phrases constitutes whatever phrases were played by the previous tflush instruction.)

Note: The **talkresume** instruction cannot be used to resume TTS play.

- **tstop(*[type.scr]*)**

  The **tstop** instruction lets the script developer stop any speech activity on the script's current channel.

The following is an example using the **tstop** instruction:

```
talk(int.MUSIC) /* Play music to the caller */
tflush(1,1) /* Do not let touch tones turn off music
and don't wait */
dbase(0, FUDB, ch.ACCOUNT_ID, 8, int.SELL_PRICE, 4)
   /* Get info from host */
tstop(1)
talk("Your account has now been credited with Lucent
Technologies stock for the price of")
tnum(int.SELL_PRICE)
```

In this example, the script wants the caller to hear music while it processes the transaction with the host computer. After this processing completes, the music is stopped, and the caller is informed of the results.

- **background("*phrase_name*",*type.src[,type.style]*)**
  background(*type.phrase,type.src[,type.style]*)

Note: A time division multiplexor (TDM) bus and an SSP or LSPS II circuit card must be installed in the system for the **background** instruction to function properly. See the maintenance book for your platform for information on installing these components.

The **background** script instruction starts and/or listens to background audio on the specified channel. The first argument is a phrase enclosed in quotation marks (" "). The phrase must match a phrase listed in the talkfile specified by the currently active **tfile** instruction. The first

argument can translate also to the index number of a phrase in the talkfile; in this case, the argument must be expressed according to the conventions of *type.src*. This syntax is similar to the **talk** instruction but it only supports one phrase rather than a phrase list. The optional *type.style* argument specifies the coding style of the speech file.

If this phrase is not playing already in the system, it is started and its audio output added to the normal voice response prompts on the current channel. Other channels may execute the same **background** instructions; the audio then is added to those channels while still playing on the first channel. When the phrase has been played, it starts again at the beginning. The phrase continues to play as long as at least one channel requires it. The **background** audio stops when all channels requesting it have dropped it. Background speech plays at a volume level that is 33 percent of the foreground speech for the SSP circuit card. Background speech on the LSPS II circuit card must be recorded at lower levels because the LSPS II circuit card does not provide gain control.

If the **background** instruction is successful, it returns a positive value in *r.0*. If the instruction is not successful, it returns a negative value in *r.0*.

The following are possible reasons the **background** instruction might fail:

- An attempt to add more than one background audio to a channel failed.
- The channel reached its limit for listen time slots (maximum seven per channel).
- No SSP or LSPS II circuit card is available.

- ~ All TDM slots are busy.
- ~ The system reached its limit on the number of **background** instructions that can be specified (MAXCHAN).
- ~ A system call failure occurred.

Below is an example of how the **background** instruction might be used in a script.

```
#define ADD 1
#define DROP 0
tfile("/speech/talk/list.cabnt")
background("begin testing",ADD)
background(201,DROP)
```

- **setalk(*type.talk*)**

  The **setalk** instruction is used to specify a new talkfile for talk instructions. The *type.talk* argument is the id of the new talkfile. After **setalk** is executed, the previous talkfile id is returned in *r.0* and can be saved for future use. The **setalk** instruction overrides the talkfile number contained in the first listfile specified in the **tfile** instruction.

- **setftalk(*ctype.src*)**

  The setftalk script instruction specifies a new prepend directory for the **fsay** and **ftalk** instructions. The argument, *ctype.src*, is the full pathname of the prepend directory. The prepend directory cannot be longer than 128 characters. When a relative pathname is given to **fsay** or **ftalk,** the sum of lengths of the prepend directory and the relative pathname cannot exceed 128 characters.

Sample Script Using Voice Output Instructions

In this example using voice output instructions, the script tells TAS to use the "stocks.pl" speech database for this transaction, then welcomes the caller to the application. The script plays the special announcements, which cannot be interrupted by touch tones, because of the **tflush** instruction. The script asks for the caller's account number and repeats for the caller what has been entered. The script tells the caller how many shares they own based on the value stored at address VOLUME.

```
MAIN:
#define VOLUME 0
#define INITIALS 4
--
--
--
tfile("stocks.pl")
talk("Welcome to our stock quote system")
```

```
talk(“A special announcement: our service will be offered
24-hours a day”)
tflush(1)
talk(“Please enter your account number”)
--
--
--
talk("Your initials are")
tchars(ch.INITIALS)
talk("You currently own")
tnum(int.VOLUME)
talk("shares of Acme Incorporated.")
--
--
--
```

Data Gathering Instructions

The data gathering instructions get information from a caller or from a stored database. This section describes the data gathering instructions and provides examples of those type of instructions. A sample script at this end of this section illustrates how these instructions might be used in an application.

- **getinput(*ctype.dst, type.number*[, *int.recognizer*[, *int.resource*]])**

  The **getinput** instruction receives information entered by a calling party using touch tones, dial pulses, or speech input.

The instruction **getinput** replaces **getdig**. Continued use of **getdig** is discouraged.

The argument *ctype.dst* is a character buffer where input data is to be copied to. The argument *type.number* indicates the maximum number of input characters to copy to *ctype.dst.* The argument *int.recognizer* is optional, it indicates the address of the integer value where the recognition type used to collect input is stored. Possible values include 0 for TT input or some positive integer indicating a recognizer such as IRD_WHOLE_WORD (see **recog_start**). The argument *int.resource* is optional, it indicates the address of an integer where the resource used to perform recognition is stored.

getinput has a 10 second default initial digit wait time for input. If the caller does not enter a digit within the allotted time period, **getinput** returns the number of digits received before the timeout occurred. Use the **tttime()** instruction to specify desired wait times.

getinput is a wait-causing instruction. Therefore, it automatically forces any pending or unfinished announcements to be played from this channel.

For information about the **getinput** instruction in relation to speech recognition, see Dial Pulse and Speech Recognition Script Instructions on page 92.

- **tttime*(type.firstdig,type.interdig)***

  The **tttime** instruction allows a script to set the time-out values for receiving touch tones. The *firstdig* argument specifies the maximum number of seconds the script should wait to receive the first touch-tone digit after executing a **getinput** instruction. The *interdig* argument specifies the maximum number of seconds to wait between two consecutive touch-tone digits.

  There is no limit to the timeout values. The default values are 10 and 10.

  The **tttime** instruction is related to the **getinput** instruction. If the *firstdig* time is exceeded, *r.0* is set to 0 and the **getinput** instruction terminates. If the *interdig* time is exceeded, *r.0* is set to the number of digits that are received, transferred to the script buffer, and the **getinput** instruction terminates.

  In the following **tttime** example, the script waits approximately ten seconds for the first touch tone and two seconds between touch tones.

  ```
  tttime(10,2)
  ```

- **setttfl(*type.flg*)**

  The **setttfl** instruction allows the script to change the way TSM handles the touch-tone buffer. Normally, TSM gets rid of any touch tones it has received for the script when the speech buffer is flushed and speech is played. The **setttfl** instruction disables the TSM action of clearing the touch-tone buffer whenever speech is played.

If the *type.flg* argument is 1, touch-tone flushing is turned on. If the **setttfl** instruction is not used, the default condition sets touch-tone flushing on.

If the *type.flg* argument is 0, touch-tone flushing is turned off and playing speech does not cause the touch-tone buffer to be cleared. Another effect of turning off touch-tone flushing is that any phrases queued in the phrase buffer are cleared if talkoff is enabled on the channel instead of being played whenever an instruction that normally causes the phrases to be played is executed. This is done because phrases that are in the buffer are assumed to be part of the prompt that the talkoff touch tones affect. With talkoff enabled, phrases that are already queued are not heard. Instead, the script advances to the appropriate point based on the touch-tone input received.

- **ttclear()**

  The **ttclear** instruction clears the touch-tone buffer. This instruction is useful for applications in which you want to throw away all typed-ahead input. The **ttclear** instruction removes any touch tones in the touch-tone buffer when the instruction is executed. The number of touch tones cleared is stored in *r.0*.

- **ttdelim(*erase-char, erase-all, delim1, delim2*)**

  The **ttdelim** instruction sets four control functions and the touch-tone keys used by the caller to perform those functions. The functions for the *erase-char* and *erase-all* arguments are defined by the system; the functions for the *delim1* and *delim2* arguments are defined by the application developer. The application developer defines the touch-tone keys used in performing all four functions.

  The system-defined functions *erase-char* and *erase-all* do not terminate the collection of touch tones initiated by the **getinput** instruction and those characters are removed from the buffer. The developer-defined functions *delim1* and *delim2* terminate the collection of touch tones and those characters remain in the buffer.

  The **ttdelim** instruction works with the **getinput** and **tttime** instructions. For example, after requesting five digits with a **getinput** instruction, normally *r.0* is set to 5 and the actual digits received are stored at the destination. Any time the **ttdelim** instruction is used, the script also has to check the received digits to determine if *delim1* or *delim2* was used.

  The touch-tone buffer is scanned for the delineators that are in effect when a **getinput** instruction is executed.

The values for the **ttdelim** arguments are shown in Table 7:

Table 7. ttdelim Arguments

Value	Meaning
-1	Function is not used (default)
0	Do not change value of current function
'c' or 'cc'	New value where *c* is: 0–9, #, * or A-D (only on extended keypad, such as an operator console)

The following functions and characters might be specified for the instruction:

```
ttdelim('#1','#*','*1','*2')
```

Characters	Meaning
#1	Erase one character
#*	Erase all characters
*1	Get operator
*2	Play help message

Script routines written by the application developer must check for *1 and *2 in the buffer. If the **ttdelim** instruction uses only one argument, then a default value must be entered for the other three arguments. An example of a **ttdelim** instruction using only the *erase-all* function is **ttdelim(-1,'#',-1,-1)**. Whenever *erase-char* and *erase-all* are used in a script, a *delim* argument is probably used to allow a caller to end touch-tone entry. This argument indicates to the **getinput** instruction that although it may have received the maximum number of digits, a caller may make a mistake and may want to erase some digits and re-enter them.

To allow for the extra digits requested by a *delim1* or *delim2* argument, the **getinput** instruction should specify more digits than it needs. For instance, if a 5-digit entry is required, but it is anticipated that a caller might enter all incorrectly and need to erase them, **getinput** would require a minimum of 7 digits to accommodate the two-digit delineator for *erase-all.*

Based on the previous arguments for the sample **ttdelim** instruction shown earlier, the **getinput** instruction would have the results given by the examples in Table 8 on page 57.

Table 8. getinput Results

Input	*r.0*	Destination	Script Action
12345	5	12345	Use 5 digits
123#*45678	5	45678	Use 5 digits
12*1	4	12*1	Transfer to operator
*1	2	*1	Transfer to operator
12*2	4	12*2	Play help message and reprompt for input

The time-outs for the system-defined functions, *erase-char* and *erase-all*, are the same. The **tttime** instruction only uses the *firstsec* argument once, but it repeatedly uses the *interdig* argument to wait the maximum amount of time specified for receiving the next digit. The application developer needs to write code to implement the functions. For example, *delim2* would need a **talk** instruction to play a help message.

- **dbase(*dip,type.mcont_field, type.dst,mbyte,type.src,nbyte*)**

  The **dbase** instruction passes information between the script and a host, a local database, or any other DIP.

  The *dip* argument specifies the DIP that is to receive the message. A DIP number or name may be used for *dip*. If *dip* is a name, it must be in the

form *"name"*. The *type.mcont_field* argument is a value sent to the DIP that the DIP uses to identify the message type and determine its next action. The *ctype.dst* argument specifies the destination script address for the data. The *mbyte* argument specifies its length. The *type.src* argument specifies the script address where data sent to the DIP is stored; the *nbyte* argument specifies its length. If *type.src* is a register, nbyte is ignored and four bytes are sent.

If the **dbase** call is successful and returns data to the script, *r.0* is set to the *mcont* value of the DIP message. If the DIP is not running, *r.0* is set to -1. After a response timeout (default value is 45 seconds), *r.0* is set to -2. To change the default value for the timeout, use the **nwitime** instruction described later in this chapter. If *nbyte* is zero (0), no information is transferred to the DIP. If *nbyte* is negative, no message is sent to the DIP, but the dbase call may wait (if *mbyte* is not negative) for a message from the DIP. If *mbyte* is negative, no return data is expected from the DIP. *r.0* is set to zero (0) and script execution continues immediately after **dbase** is executed.

In the following example, the **dbase** instruction tells the script to send ch.INFO_TO_HOST (nine bytes) to the host. The DIP "Bankdip" processes the information to the host based on the action defined by ACCOUNT_BAL and stores the result in ch.INFO_FROM_HOST (up to 55 bytes).

```
dbase("Bankdip",ACCOUNT_BAL,ch.INFO_FROM_HOST,
55,ch.INFO_TO_HOST,9)
```

- **dipterm(*dip*[,*flag*])**

  The **dipterm** instruction specifies to TSM that a DIP receives a termination message when the script terminates. A DIP number or name may be used for *dip*. The **dipterm** instruction may be called repeatedly with different DIP numbers or names. The termination message goes to all DIPs specified.

  The optional *flag* may be used to turn off a dipterm setting. Valid values for *flag* are 1 and 0. If *flag* is 1 (the default), **dipterm** is set for the *dip*. If *flag* is 0, **dipterm** is reset for *dip*.

  The following **dipterm** instruction example tells TSM that DIP0 will receive a termination message when the script completes.

  ```
  dipterm(0)
  ```

  For additional information about the **dipterm** instruction (such as reasons for termination reported in the header file **tsm_dip.h**), see Appendix B, Summary of TAS Script Instructions.

Sample Script Using Data Gathering Instructions

In this example, the script instructs the system to prompt the caller for a passcode. The script waits for the caller to enter three touch tones and stores them in ch.PASSCODE. The script then tells the system to send ch.PASSCODE (3 bytes) to the host. The host performs ACCOUNT_BAL processing, then returns up to 55 bytes of data. That data is stored in ch.INFO_FROM_HOST.

```
#include HOST_HEADER.h
#define PASSCODE 0
#define INFO_FROM HOST 4

MAIN:
--
--
talk("Enter your 3-digit pass code.")
getinput(ch.PASSCODE,3)
--
--
--
dbase(0,ACCOUNT_BAL,ch.INFO_FROM_HOST,55,ch.PASSCODE,3)
--
--
--
```

Data Manipulation Instructions

The data manipulation instructions perform arithmetic data functions and also change the contents of memory. Following the list of the instructions and descriptions of each are two sample scripts which illustrate how the instructions might be used in an application. For additional information about each of these instructions, see Appendix B, Summary of TAS Script Instructions.

- **and(*type.dst,type.src*)**

  The **and** instruction implements bitwise AND operations on the *type.dst* and *type.src* arguments, allowing scripts to decode or encode bit flags stored in a single integer. The result is stored in *type.dst*.

- **atoi(*type.dst,ctype.src*)**

  The **atoi** instruction converts a null terminated character string at the source to an integer value and stores that value at the destination. If an error occurs, a 0 is returned at the destination.

- **decr(*type.dst,type.src*)**

  The **decr** instruction decrements the destination value by the source value.

- **div(*type.dst,type.src*)**

  The **div** instruction divides the destination value by the source value. The integer quotient is stored in *type.dst*. The remainder is discarded. The **div**

instruction returns a value of 0 in *r.0* if no error occurred. If division by 0 is done and a -1 value is returned in *r.0*, the result is set to the largest positive or negative integer, depending on whether *type.dst* was positive or negative originally.

- **dtitos(*type.src,type.dst*)**

  The **dtitos** instruction converts date and time data from internal UnixWare system form to “tm” structure form. The *type.src* argument should contain a number representing the UnixWare system internal representation of time (number of seconds since 00:00:00 GMT January 1, 1970). It is recommended that the integer type be used for this argument. The resulting “tm” structure (9-integer structure defined in ctime(3C) in the *UnixWare Operating System API Reference*) is put in *type.dst* (that is, *type.dst* defines a starting address for the result).

  The **dtitos** instruction returns 0 in script r.0 if the conversion is successful. A -2 is returned in r.0 if TSM could not allocate enough space in script memory to store the result.

- **dtstoi(*type.src,type.dst*)**

  The **dtstoi** instruction converts date and time data from “tm” structure to internal UnixWare system form. The “tm” structure is specified by the *type.src* argument. The result is placed in *type.dst*. It is recommended that the *type.dst* argument use type *integer* to guarantee that the correct value is received. This instruction is the complement to the **dtitos()** instruction.

The **dtstoi()** instruction returns 0 in script r.0 if the conversion is successful. A value of -1 is returned in r.0 if the "tm" structure indicated by *type.src* contains incorrect values or is at a location outside the script data area.

- **incr(*type.dst,type.src*)**

  The **incr** instruction increments the destination value by the source value.

- **itoa(*ctype.dst,type.src*)**

  The **itoa** instruction converts a numeric source value into a null terminated character string stored starting at the destination address.

- **load(*type.dst,type.src*)**
  load(*ctype.dst,ctype.src*)

  The **load** instruction converts the source value to the data type of the destination and stores it at the destination.

- **mul(*type.dst,type.src)***

  The **mul** instruction multiplies the destination value by the source value. The product is stored in *type.dst.* Overflow is not checked; multiplying large values may result in a negative number.

- **not(*type.dst*)**

  The **not** instruction performs a 1's complement operation on the *type.dst* argument, allowing scripts to decode or encode bit flags stored in a single integer.

- **or(*type.dst,type.src*)**

  The **or** instruction implements bitwise OR operations on the *type.dst* and *type.src* arguments, allowing scripts to decode or encode bit flags stored in a single integer. The result is stored in *type.dst*.

Sample Scripts Using Data Manipulation Instructions

The following are two sample scripts using the data manipulation instructions as they might be used in an application. The second example uses several instructions introduced later in this chapter.

In the following example, the script asks the caller to enter the number of widgets they want to order, then waits for three touch tones and stores them in ch.QUANTITY. The script then converts the characters in ch.QUANTITY to an integer and stores it in r.1. The script tells the caller how many widgets they ordered based on the integer in r.1. Using the **mul** instruction and r.1, the script multiplies the integer in r.1 by 5 (5) to get the total cost of the order. The script then tells the caller the total cost of the order.

```
MAIN:
      talk("Enter the number of widgets you wish to order")
      getinput(ch.QUANTITY,3)
      atoi(r.1,ch.QUANTITY)
      talk("You have ordered")
      tnum(r.1) /*spoken as a number, not a string of digits*/
      talk("widgets")
      talk("at a cost of $5 each for a total cost of")
      mul(r.1,5)
      tnum(r.1,'f')
      talk("dollars.")
```

In the following example, the script sets the value of r.1 to 0, then asks the caller to enter their password. The script waits for four touch tones, stores them in ch.PASSWORD, then compares ch.PASSWORD with ch.GOOD. If they match, the script continues. If they do not match, as this example illustrates, the script jumps to the retry instructions where it tells the caller that the password is invalid. The script increments r.1 by 1. If r.1 equals 3, the script jumps to the *good-bye* subroutine. If r.1 is not equal to 3, the script asks the caller to re-enter the password. The script then goes back to the validate subroutine. In this example, the caller can enter an invalid password up to three times before the script terminates in good-bye.

```
start:
load(r.1,0)
talk("Enter your password.")

validate:
getinput(ch.PASSWORD,4)
strcmp(ch.PASSWORD,ch.GOOD)
jmp(r.0 !=0,retry)
--
--
retry:
talk("I’m sorry, that was an invalid password")
incr(r.1,1)
jmp(r.1 == 3,good-bye)
talk("Please re-enter your password.")
goto(validate)
good-bye:
talk(“Goodbye”)
quit()
```

String Instructions

The following script instructions recognize the use of the double-quote syntax to indicate a literal, null-terminated ASCII character string. Although the **talk** instruction also uses a double-quote syntax, the meaning is different; it implies a talkfile search for phrases that match the string.

- **strcmp(*ctype.src*,*ctype.src*[,*type.len*])**

  The **strcmp** instruction allows a script to compare two character strings. The *ctype.src* arguments can be either an address or a literal string. The results of the comparison are returned in r.0. The return value is interpreted as follows:

If *r.0* is …	Then …
=0	The strings are equal (exactly the same)
<0	The first string is lexicographically less than the second string
>0	The first string is lexicographically greater than the second string

  If the optional *type.len* argument is used, the comparison is limited to the number of characters specified by that argument.

Below are two examples of the **strcmp** instruction. In the first example, the **strcmp** instruction returns a value less than 0 because “abc” is lexicographically less than “abx.” In other words, the string “abc” appears before the string “abx” in an alphabetical listing. In the second example, the return value is greater than 0 because “abcd” appears before “abx” in an alphabetical listing even though “abcd” has more characters than “abx.”

```
strcmp(“abc”,“abx”)
strcmp(“abx”,“abcd”)
```

Note: Capital letters are always lexicographically less than lowercase letters and numbers are always lexicographically less than letters.

- **strcpy(*ctype.dst*,*ctype.src*[,*type.len*])**

  The **strcpy** instruction allows a script to copy a source string to a specified destination. The *ctype.dst* argument specifies the destination address to which the source string (including the terminating null character), specified by the second argument, is copied. The first argument must be an address. The second *ctype.src* argument specifies the source string to be copied. This argument may be a literal string or the address at which the first character of the string is located.

  If the optional *type.len* argument is used, the **strcpy** instruction copies, at most, the number of characters specified by that argument. The result may or may not be null terminated, depending on whether a null character was copied before the *type.len* character limit was reached.

Below are examples of the **strcpy** instruction. In the first example, the string ch.ORIGINAL is copied to the destination ch.COPY. In the second example, the string “Welcome” is copied to the destination ch.COPY.

```
strcpy(ch.COPY,ch.ORIGINAL)
strcpy(ch.COPY,“Welcome”)
```

- **strlen(*ctype.src*)**

  The **strlen** instruction computes the length of the string specified by the *ctype.src* argument. The *type.src* argument can be a literal string or the location of a string. The length of the string (that is, the number of characters in the string) is returned in r.0.

  In the following **strlen** instruction example, **getdig** looks for nine touch tones and stores them in ch.SOCIAL_S_NUM. The **strlen** instruction computes the length of the string stored in ch.SOCIAL_S_NUM and stores the value in r.0. Then the **jmp** instruction looks at the value in r.0 and, if it is less than nine, goes to the code at invalid_num.

  ```
  getdig(0,ch.SOCIAL_S_NUM,9)
  strlen(ch.SOCIAL_S_NUM)
  jmp(r.0< 9,invalid_num)
  ```

Flow Control Instructions

Flow control instructions determine the order in which the instructions are executed. Each instruction is listed with a brief description. An example of a script using these instructions follows the descriptions.

- **case(*type.src*,*type.src*,<*subroutine_label*>,<*goto_label*>)**
  case(*type.src*,*type.src*,<*subroutine_label()*>,<*goto_label*>)
  case(*type.src*,*type.src*,<*subroutine_label*(*type.src*)>,<*goto_label*>)
  case(*type.src*,*type.src*,<*subroutine_label(type.src,type.src)*>,
  <*goto_label*>)

  The **case** instruction provides a conditional subroutine call that compares two source values. If the source values are equal, the subroutine is called, and on return, execution continues at the *goto_label* address. If they are not equal, the statement does nothing. If the *subroutine_label* is a negative number, no subroutine call is made and execution continued at the *goto_label*. If the goto_label is negative, execution continues with the next instruction.

  Subroutine calls invoked in a case statement behave like other subroutine calls (that is, with arguments allowed and register values saved on the stack).

- **event(*event_type*[, *subroutine_label*])**
  event(*event_type*[, *type.offset*])

  The event instruction allows script execution to continue after certain events occur, such as when the caller hangs up or the script detects another external event. The event script instruction causes a jump to the *subroutine_label* given when events defined by the *event_type* argument occur. The event types are defined in the **/att/msgipc/tsm_dip.h** header file.

  If valid arguments are passed, the event instruction returns an integer offset in r.0. This offset is the value of the previous *subroutine_label* (if any) used for the event. It may be saved and used later as the *type.offset* argument to the **event** instruction to reset the *subroutine_label* back to its previous value. (This is useful for external script functions which need to handle events and want to restore their disposition to whatever the calling script had set before returning.)

  If *event_type* is not valid or *type.offset* is larger than the text space of the script, a value of -3 is returned by the event instruction.

  A negative value for *type.offset* may be used to set no subroutine label for an event, causing the default action to be taken when the event occurs (see below). If no *subroutine_label* or offset is given, the **event** instruction returns in r.0 the value of the *subroutine_label* currently being used (or -1 if none) without changing the disposition for the event.

The event types are described briefly below. See Appendix B, Summary of TAS Script Instructions, for more information about the event instruction and event types.

- EHANGUP specifies a hangup event. This event is triggered when dial tone, no loop current, disconnect, or glare conditions are detected on the channel.
- EDIALTONE and ESTUTTERDT specify a dial tone event. These are special cases of the EHANGUP event. Normally, EHANGUP is triggered when dial tone or stutter dial tone is detected (and the script is not expecting dial tone). EDIALTONE and ESTUTTERDT are used to treat dial tone detection separately from EHANGUP.
- ESOFTDISC specifies a soft disconnect event. This event is triggered by sending a SOFT_DISC message to TSM from a DIP. If an event subroutine is set, it receives the following values when the event occurs:

  r.0 Event type (ESOFTDISC)

  r.1 Value from arg[1] of SOFT_DISC message

  r.2 Value from arg[2] of SOFT_DISC message

  r.3 Number of the DIP that sent the SOFT_DISC message

~ EDIPINT specifies a DIP interrupt event. This event may be triggered by sending a DIP_INT message from TSM to a DIP. If an event subroutine is set, it receives the following values when the event occurs:

r.0 Event type (EDIPINT)

r.1 Value from arg[1] of DIP_INT message

r.2 Value from arg[2] of DIP_INT message

r.3 Number of the DIP that sent the DIP_INT message

~ ETTREC specifies a touch-tone received event. This event can be used to allow a **dbase**, **sleep)**, **tflush,** or **tic** instruction to be interrupted if a touch tone is received while they are being executed.

Note: The **tflush** instruction is only interrupted if its first argument is 1 (that is, talkoff is disabled).

If an event subroutine is set, it receives the following values when the event occurs:

r.0 Event type (ETTREC)

r.1 Touch tone character that caused the interrupt

r.2 Number of touch tones received since last **getdig** or **ttclear**

r.3 Instruction interrupted: 't' - **tflush**, 's' - **sleep**, 'd' - **dbase**, 'i' - **tic**

If no event subroutine is set for ETTREC, the instructions are not interrupted by touch tones.

~ EANSSUP specifies an answer supervision event. This event is triggered when answer supervision (rather than speech energy detection) is detected as available for a any telephony interface. This includes T1 (E&M), E1 (CAS), and PRI. It also includes Tip/Ring and LSE1/LST1 when the system is connected to a DEFINITY ECS or compatible switch and the DTMF Feedback to VRU feature is available.

- **exec(*ctype.script*[,*type.data*,*type.nbytes*][,*exitval*])**

  The **exec** instruction allows a script to execute another script or IRAPI application.

The *ctype.script* argument is the name of the script to be executed. The *type.data* and *type.nbytes* optional arguments are used to pass a block of data to the new application. The *type.data* argument specifies the location of the data and the *type.nbytes* argument specifies the size, in bytes, of that data. If *type.data* is a register or immediate type, *type.nbytes* is ignored and a size of an integer (4 bytes) is assumed. These two arguments work like the last two arguments of the **dbase** instruction.

The *exitval* argument is an optional exit value used when the parent script is terminated before the new child script is run. It is passed to a DIP specified by the **dipterm** instruction in the same way as the argument to the **quit** instruction and may be specified without using either *type.data* or *type.nbytes.* If no exitval is given, -1 is used by default. See Appendix B, Summary of TAS Script Instructions, for more information on the **exec** instruction (see also: **subprog**).

- **execu(*ctype.script*[,*type.data*,*type.nbytes*][,*exitval*])**

  The **execu** instruction has the same format and functionality as **exec**. Using **execu** instead of **exec**, however, causes the new script to inherit, the user data space of the parent script intact. Essentially, this feature allows a script to pass all user data to the new script. For this to be useful, however, the new script must have its data defined in the same way as the parent script (that is, structures, variables, etc., must be defined for the same locations). The data definition of the new script is used to overlay the actual data of the parent script (see also: **subprog**).

- **goto(*<label>*)**

  The **goto** instruction is an unconditional jump to the instruction indicated by the *label*.

- **ibrl(*type.dst*,*type.src*,*<label>*)**

  The **ibrl** instruction, which means increment and branch if less, determines if another pass should be made through a loop. The **ibrl** instruction normally is placed at the end of the loop. The destination value is incremented by one and then compared to the source value. If the destination is less than the source value, a jump to the labeled instruction is executed and the loop is repeated. If the destination is greater than or equal to the source, the next instruction is executed.

- **jmp(*type.src rel_op type.src*,*<label>*)**

  The **jmp** instruction is a conditional jump to the labeled instruction. The *rel_op* argument compares the values of the two source operands. If the condition is true, a jump to the labeled instruction is executed; if false, the statement does nothing. The *rel_op* has six C-style operators:

If *rel_op* is . . .	Then . . .
==	Equal
!=	Not equal

1 of 2

If *rel_op* is . . .	Then . . .
<	Less than
<=	Less than or equal to
>	Greater than
<=	Greater than or equal to

2 of 2

- **<*label*>:**

  A **label** definition assigns a label to a program segment. It is not the same as **label( )** (described below), which is a subroutine call.

- **<*label*>([*type.src*] [,*type.src*]])**

  The label instruction is used to call a subroutine found at the address indicated by the ***<label>:*** of the segment. A return address and the values in all registers except r.0 are saved on a subroutine stack. An optional first argument is stored in r.3 and an optional second argument is stored in r.2 for use by the called subroutine; otherwise, the registers are left unchanged.

- **nap(*type.src*)**

  The **nap** instruction makes the script do nothing for the specified number of centiseconds (hundredths of a second). See **sleep**.

- **nwitime(*type.src*)**

  The **nwitime** (next wait instruction time) instruction sets the maximum amount of time the script waits for the completion of the next instance of the following wait-causing instructions. The *type.src* argument specifies the number of seconds to wait.

  - background
  - dbase
  - phreserve
  - sr_talkoff
  - tic
  - tstop

- **quit([*exitval*])**

  The **quit** instruction terminates the script voluntarily. An optional parameter is passed to a DIP specified in a **dipterm** instruction. This *exitval* is also returned to the parent script in register 1 (r.1) if the script was executed with the **subprog()** instruction.

- **rts()**

  The **rts** instruction is the mechanism for returning from a subroutine call. The saved values for all script registers except r.0 are restored.

- **scrinst(*[ctype.script]*)**

  The **scrinst** instruction enables an application script to find out how many instances of a script are running currently on the system. Based on the value returned by this instruction, the script may choose to prohibit execution of another instance of the script (via the **exec** instruction) or the script may quit if it is performing a check on itself and has exceeded the limit.

  The *ctype.script* optional argument is the script, or service, name. If no script name is given, the script executing the instruction is assumed. This instruction sets the value of r.0 to the number of instances of the given script at the time the instruction is invoked.

  There are several possible uses of **scrinst** based on the ways in which a script may be started:

  ~ Incoming call — It is suggested that the method of limiting the number of scripts started with an incoming call be left as it is. That is, do not assign a service to a number of channels greater than the desired limit. If the number of channels assigned to a script exceeds the limit, a script still may check the instance count as its first task and quit before answering the call if the instances exceed the limit.

  ~ **exec** — The **exec** script instruction is the primary means by which an instance limit may be exceeded. Therefore, any application script concerned about running too many instances of another script should use **scrinst** for that script before using **exec**.

In this case, it is important to avoid a wait condition in the interval between **scrinst** and **exec**. This could cause other scripts running simultaneously that are performing the same test to receive identical results from **scrinst** before any of them perform the **exec** instruction. Use **tflush** before **scrinst** to play any speech that is queued. If **tflush** is not used, the **exec** instruction causes the speech to play and the script waits for the play to complete before executing the **exec** instruction.

~ Soft seizure — Scripts started by a soft seizure request from a DIP may use **scrinst** to check themselves against an instance limit as their first task, similar to the way **scrinst** may be used if the script is started by an incoming call. If the script determines that it cannot continue, it may signal the DIP that started it by using the **dipterm** instruction and calling **quit** with a specific value that the DIP may check.

- **sleep(*type.src*)**

  The **sleep** instruction makes the script do nothing for the number of seconds specified by the argument. See **nap**.

- **subprog(*ctype.appname*[, *type.data, type.nbytes*])**

  The **subprog** instruction allows a script to execute another TAS script or IRAPI application as a subprogram, and then return to the parent script with data from the subprogram.

The *ctype.appname* argument is the name of the script to be executed. The *type.data* and *type.nbytes* optional arguments are used to pass a block of data to the new application. The *type.data* argument specifies the location of the data and the *type.nbytes* argument specifies the size, in bytes, of that data. If *type.data* is a register or immediate type, *type.nbytes* is ignored and a size of an integer (4 bytes) is assumed. These two arguments work like the last two arguments of the **dbase** instruction.

subprog returns values in both register 0 (*r.0*) and register 1 *(r.1*). *r.0* is 0 if **subprog** was successful. It contains a negative value to indicate failure. If successful, **subprog** also returns in *r.1* whatever value was passed to the **quit** instruction by a called TSM script application, or whatever value was passed to **irReturn** by a called IRAPI application. See Appendix B, Summary of TAS Script Instructions, for more information on the **subprog** instruction (see also: **exec**).

Sample Script Using Flow Control Instructions

The following is an example of a script using the flow control instructions.

```
#define DECIDE 0
#define COUNTER 2
INTRO:
talk("Welcome to our company")
load(r.1,0) /*initialize loop counter to 0*/
```

```
start:
talk("To speak to an operator, enter 1")
talk("To hear your account balance, enter 2")
getinput(ch.DECIDE,1)
case(ch.DECIDE,'1',OPERATOR,continue)
case(ch.DECIDE,'2',ACCT_BAL,continue)
--
--
--

OPERATOR:
talk("Please hold, an operator will be with you shortly")
--
--
(code to dial operator, transfer call)
--
--
(call returns)
rts()

ACCT_BAL:
nwitime(20) /*maximum seconds to wait for host
confirmation*/
--
--
```

```
(query host)
--
--
talk("Your account balance is")
tnum(int.FIVE,'f')
rts()

continue:
ibrl(COUNTER,r.1,start)
talk("Thank you for calling")
quit()
```

In this example, the instructions first define DECIDE as 0 and COUNTER as 2. The script then welcomes the caller to the system and initializes r.1 as containing 0. The script asks the caller to enter 1 to talk to an operator and 2 to hear the account balance. The **getinput** instruction tells the script to wait for one touch tone and store it in ch.DECIDE. The **case** instructions tell the script that if the caller enters '1', to go to the OPERATOR subroutine, then to the continue code and if the caller enters '2', to go to the ACCT_AL subroutine, then to the continue code. Regardless of what the caller enters, script execution continues with the next instruction.

The OPERATOR subroutine would contain code telling the script to dial out to an operator and transfer the call. (This code is omitted from this example to make it simple, as shown by the use of --). When the caller has finished talking to the operator, the script continues with the next instruction.

The ACCT_BAL subroutine tells the script to wait a maximum of 20 seconds for the host information requested (**nwitime** instruction). The call to the host is not included here to keep the example simple. After the host has returned the information, the script tells the caller what the account balance is based on the value in the **tnum** instruction. The script then continues with the next instruction.

The **ibrl** instruction tells the script to compare COUNTER (which was set at 2 at the beginning of the script) with the value in r.1. If r.1 is less than COUNTER, the script returns to the start code. If r.1 is equal to COUNTER, the script executes the next instruction. The script then thanks the caller for calling and quits, voluntarily ending the transaction.

Voice Coding Instructions

Voice coding instructions provide script facilities for adding or removing phrase numbers to or from a selected speech file. These instructions also store speech within these or previously-defined phrase allocations. These facilities may be used, with suitable script prompts, to record user voice or touch-tone messages.

The feature of ending the voice coding session by pressing a touch-tone key (referred to as talkoff) can be disabled using **tflush(1)** before the **vc** instruction. This allows the user to encode the touch tones as well as the speech. See Voice Output Instructions on page 34 for details about the **tflush** instruction.

The voice coding instructions are described below. Following these is an example of a script for voice coding and play.

- **phreserve(*type.phrase*,*type.talk*,*type.time*,*type.style*)**

  The **phreserve** instruction creates an area in a talkfile that is used to store a phrase. This phrase is later encoded by the **vc** instruction. The arguments for the **phreserve** instruction are:

  ~ The *type.phrase* argument specifies the phrase id of the phrase to be created (valid range is 1–65,535).

  ~ The *type.talk* argument specifies the talkfile id of the talkfile where the phrase is stored (valid range is 1–16,383).

  ~ The *type.time* argument specifies the amount of space, or time (in seconds), to be reserved for a phrase in the talkfile.

  ~ The *type.style* argument specifies the coding style and rate to be used. Valid coding styles and rates are defined in the header file **codestyle.h** in the directory **/att/include**. This file should be included in the script by an **include** instruction. If the style specified is not valid, the **phreserve** instruction fails.

  Valid coding designations are as follows in Table 9 on page 86:

Table 9. Coding Designations

	Coding Style	Description
SSP	ADPCM32	Adaptive differential pulse code modulation at 32 Kbps
	ADPCM16	ADPCM at 16 Kbps
	CELP16	code excited linear prediction at 16 Kbps
	SBC24	Sub-band coding at 24 Kbps
	SBC16	Sub-band coding at 16 Kbps
LSPS II	PCM (Mu-law)	PCM coding at 64 Kbps
	OKI ADPCM	OKI ADPCM at 32 Kbps

If *type.phrase* is -1, the system assigns a phrase id and returns this id in r.1. The phrase id can be used to reference the phrase (for example, in a **talk** instruction) once it has been coded and stored in the talkfile by the **vc** instruction. If *type.talk* is -1, the system selects the default value 255 for the talkfile and returns the id of the selected talkfile in r.0.

Note: If there are two **phreserve** instructions, there must be a **vc** instruction between them or the second **phreserve** instruction will fail.

When both *type.talk* and *type.phrase* are -1, both a phrase id and talkfile id are chosen by the system and returned in r.1 and r.0, respectively. These selections start with the largest previously unassigned phrase number of talkfile 255. Subsequent phrase selections fill unused phrases of talkfile 255 toward phrase 0. Since r.0 and r.1 can be used implicitly to store talkfile and/or phrase ids, the script writer must take care to save the contents of these registers before the **phreserve** command is executed.

If *type.phrase* matches the phrase id in the specified talkfile, the existing phrase is replaced by the new phrase. The values 0 and -1 for the *type.time* argument indicate that the **phreserve** instruction should not allocate any space. If enough space is available to store the phrase when coding ends, the phrase is stored. If there is not enough space, an error message is issued from the **vc** instruction.

If the instruction is completed successfully, the return values are:

~ *r.0* = talkfile id

~ *r.1* = phrase id

If the instruction is not completed successfully, the return value in r.0 is negative.

- **phremove(*type.phrase*,*type.talk*)**

  The **phremove** instruction removes the phrase specified by the *type.phrase* argument from the talkfile specified by the *type.talk* argument. The valid values for *type.phrase* are 1–65,535. The valid

values for *type.talk* are 1–16,383. *Type.phrase* must be a valid phrase id. *Type.talk* may have the value -1. If *type.talk* is -1, then the talkfile id used is the current talkfile.

If the **phremove** instruction is successful, it returns the phrase id of the phrase removed in r.0. If the instruction is not successful, it returns a negative value in r.0.

- **vc**(***flag,type.time,type.rate*[*,wait_flag*]**)**

  The **vc** instruction codes speech into a phrase in a talkfile.

  For the first argument, 'b' (for begin coding) is accepted. Another character value, 'p' (for prompt) may be used with the SSP circuit card to play a short beep just before voice coding starts. Note that this beep must be included in the prompt, either as a separate phrase or as part of the phrase.

Note: Gain control does not work for the LSPS II circuit card using this instruction.

The *type.time* argument specifies the maximum duration, in seconds, of the coding session. A value 'n' for *type.time* specifies a coding session lasting up to 'n' seconds. A value of -1 or 0 for *type.time* specifies the default maximum duration of 45 seconds. Coding can be terminated at any time by entering a touch tone.

The *type.rate* argument specifies the coding rate in kilobytes per second (Kbps). If the value given for this argument is not a valid rate or type, the instruction fails.

The feature of ending the voice coding session by entering a touch tone (referred to as talkoff) can be disabled using **tflush(1)** before the **vc** instruction. This allows the user to encode the touch tones as well as the speech. See Voice Output Instructions on page 34 for details on the **tflush** instruction.

The default value for the optional *wait_flag* argument is 1, which causes **vc()** to return when voice coding is complete. If this argument is used with a value of 0, **vc()** will return immediately after voice coding has started, allowing the script to execute more instructions while doing voice coding. This is useful for doing voice coding and speech recognition (with the **getinput()** instruction) simultaneously. Note that barge-in cannot be used during simultaneous recognition and coding. Also, recognition and coding should not occur on the same SSP circuit card. When voice coding is started without waiting for completion, **vc()** returns a value of 0 in register 0 (*r.0*). Voice coding must be stopped at a later time by the **tstop()** instruction with the optional *wait_flag* argument set to1 to get the return values (*r.0, r.1*, and *r.2*) from the completed voice coding.

If the **vc** instruction is successfully completed, it returns the phrase id in r.0. If the **vc** instruction is not completed successfully, it returns -1 in r.0. If the **vc** instruction recorded nothing because the initial silence timeout was exceeded (see vctime), it returns -2 in r.0. r.1 contains the recorded

message length in seconds (this should be 0 if r.0 is negative). r.2 is set to 1 if voice coding completed normally, 2 if coding was terminated by a touch tone (talkoff), and 3 if coding was terminated due to silence detection, that is, the intermediate silence timeout was exceeded (see **vctime**).

- **vctime(*type.src,type.src*)**

  The **vctime** instruction allows the application developer to set silence timeouts. The first *type.src* argument contains the value for the initial silence timeout. The second *type.src* argument contains the value for the interword silence timeout. The maximum timeout is 30 seconds.

  The values for the *type.src* arguments and the effect on the timeout are given below:

Value	Effective Timeout Value
X > 0	X becomes the timeout value
X = 0	Timeout is turned off
X < 0	Timeout is set to default value (5 seconds)

  This instruction does not give a return value to indicate success or failure.

Sample Script Using Voice Coding Instructions

The following example illustrates how the script instructions used for voice coding work together. The example script is a code segment which:

- Prompts the caller for a talkfile
- Creates phrase 200 in the talkfile the caller specified
- Codes the speech obtained from the caller
- Plays the phrase just coded
- Removes the phrase from the talkfile

Note that in this example, return codes from instructions such as **phreserve**, **vc**, and **phremove** are not checked by the script. These checks were not shown in order to make this example as simple as possible. Normally, all return codes should be examined so that errors can be detected. The discussion for each instruction describes the details pertinent to the return codes for that instruction.

```
#include "/att/include/codestyle.h"
tic('a') /*answer the phone*/
tfile("list.example")
talk("enter talkfile")
getinput(ch.TALK,2) /*get talkfile id*/
atoi (int.TFILE,ch.TALK) /*convert TT number to integer value*/
phreserve(200,int.TFILE,100,ADPCM32) /*create phrase 200*/
vctime(5,10) /*initial timeout 5 seconds*/
     /*interword timeout 10 seconds*/
vc('b',100,ADPCM32) /*begin coding*/
```

```
talk("the new phrase is")
setalk(int.TALK) /*change talkfiles*/
load(ch.OLD,r.0) /*save old talkfile id)*/
talk(int.200) /*play phrase just coded)*/
setalk(int.OLD) /*change back to old talkfile*/
talk("now removing phrase")
phremove(200,int.TFILE) /*remove phrase just created*/
quit()
```

Dial Pulse and Speech Recognition Script Instructions

Note: Dial Pulse Recognition (DPR) is only available on the SSP circuit card.

DPR, WholeWord speech recognition, and FlexWord speech recognition each use a recognizer, a recognition type (or wordlist for FlexWord recognition), and resources on the SSP or LSPS II circuit card. The same TAS instructions apply to all of these recognizers. Two important exceptions are:

- WholeWord recognition is the only one (other than touch tones) that allows the caller to interrupt the prompt message (barge-in) with the **sr_talkoff** instruction.
- Dial Pulse Recognition is the only one that requires training of the system with the **recog_init** instruction.

You can use DPR simultaneously with WholeWord recognition or FlexWord recognition. For each Prompt & Collect, either a WholeWord recognition type or a FlexWord wordlist must be selected. See *UCS 1000 R4.2 Speech Development, Recognition, and Processing,* 585-313-212.

Use the following TAS script instructions for including DPR, WholeWord recognition, and FlexWord recognition in new application scripts:

- **getinput**

  This script instruction receives touch-tone, dial pulse, or spoken input from a caller.

- **sr_talkoff**

  This script instruction enables/disables barge-in during a speech recognition prompt.

- r**ecog_cntl**

  This script instruction disables or enables a recognizer.

- **recog_init**

  This script instruction initializes a recognizer. It is used to initiate training parameters on Dial Pulse Recognition (DPR).

- **recog_start**

  This script instruction queues a recognizer for starting.

- **recog_stop**

  This script instruction removes a recognizer from the queue.

- **resource_alloc**

  This script instruction allocates or frees licensed system resources.

Details for these TAS script instructions, including arguments, return values, and examples, are included in Appendix B, Summary of TAS Script Instructions.

If your system includes the optional Intuity™ Feature Test Script Package (ftst), you can look at dpr_tst.t and asr_tst.t in the **/att/trans/sb** file for new code examples.

The following TAS script instructions were used for WholeWord and FlexWord speech recognition prior to this release for the UCS 1000 R4.2. Those TAS script instructions are supported for backward compatibility, but should not be used for developing new application scripts. Refer to Summary of TAS Script Instructions on page 423 for additional information about these instructions.

- **getdig** (replaced by **getinput**)
- **sp_alloc** (replaced by the **resource_alloc**)

Text-to-Speech Script Instructions

- **say (*ctype.src*)**

  The **say** instruction is used by the Text-to-Speech (TTS) feature to direct the system to speak ASCII text stored in a buffer. The argument *ctype.src* is the ASCII text string to be spoken. The script may pass text as a literally quoted string or the contents of a null-terminated field (for example, previously populated with a call to the **dbase** instruction). The maximum length of a literal string is 2048 characters.

  Say is similar to the talk instruction used for phrases of coded speech. The text passed to say is stored in a buffer that holds up to 2048 bytes of text. This buffer is flushed and the text is played when the buffer is full and another **say** instruction is executed or when any wait-causing instruction is executed.

  The **tflush** instruction may be used to flush the text-to-speech buffer and cause the text to play. The first two arguments to tflush (the *must_hear_flag* and the *wait_indicator*) have the same effect for TTS as for coded speech. (The third argument to **tflush**, the *remember_flag*, is not used for TTS.) That is, the first argument may be used to disable talkoff and the second may be used to play speech and to continue the script without waiting for the play to complete. Normally, TSM waits for a TTS play to complete before going to the next instruction. *Spinning off* a TTS play, then executing **dbase** to get the next block of text while the first block is playing avoids a delay in play between the two blocks of text. Scripts may continue executing alternate **say**, **tflush**, and **dbase** calls in this manner until all the text from a DIP is passed to **say** to be played.

The **say** instruction returns one of following values in script register 0 (*r.0*) one of the values in Table 10.

Table 10. Return Values for the say Instruction

Return Value	Return Explanation
0	The instruction completed successfully
-1	The **say** instruction failed. This happens if the text passed to **say** did not fit into one TTS buffer (2048 bytes).

As with coded speech, any TTS being played stops when the script that caused it terminates or executes a **tstop** instruction.

- **fsay(ctype.src)**

  The **fsay** instruction is similar to the say instruction, however it instructs the system to speak ASCII text stored in a file instead of a buffer.

  The *ctype.src* argument specifies the fully qualified or relative pathname of a file of ASCII text to be spoken. The script may pass the pathname as a literally quoted string, or the contents of a null-terminated field. The maximum length of the pathname is 128 characters. There is no limit to the size of the file itself.

Note: The **fsay** instruction should be used for speaking large blocks of text instead of using ttsdip.

The **tflush** instruction may be used to flush the TTS speech and cause the text to be spoken. The first two arguments to **tflush** (the must_hear_flag and the wait_indicator) have the same effect for text-to-speech as for coded speech. (The third argument to **tflush**, (the remember_flag, is not used for TTS). That is, the first argfument may be used to disable "talkoff" and the second may be used to play speech and to continue the script without waiting for the play to complete. Normally, TSM waits for a TTS play to complete before going on to the next instruction.

The **fsay** instruction returns one of the following values in script register 0 (r.0) (Table 11 on page 98).

Table 11. Return values for the fsay instructions

Return Value	Return Explanation
0	The fsay instruction completed successfully
-1	The fsay instruction failed to queue the request
-2	One of the following as occured: • The length of the full pathname passed the fsay instruction was too large • A relative pathname was passed in and the setfalk instruction was not previously called to set the prepend directory • A relative pathname was passed in and the length of the prepend directory plus the length of the relative pathname was too large

Network Interface Instructions

The network interface instruction **tic** is described below.

tic

- **tic('C', ctype.dialstr, type.rings)**
- **tic('D', ctype.dialstr)**
- **tic('F')**
- **tic('O', ctype.dialstr)**
- **tic('W', type.rings)**
- **tic('a')**
- **tic('d', ctype.dialstr)**
- **tic('f')**
- **tic('h')**
- **tic('o', ctype.dialstr)**
- **tic('w', type.rings)**

The **tic** instruction provides the script with control functions for the telephone interface line (channel) that the script is currently using. The function that the **tic** instruction performs depends on the value of its first argument. These argument values and their corresponding functions are listed below.

The **tic** instruction uses script registers 0 (*r.0*) and 1 (*r.1*) to return a result. This result may differ according to whether the script is using an E1, T1, or PRI channel. Where such variations exist, they are noted below.

- *C* — Call a number and wait for the disposition. Dial *ctype.dialstr*; turn on speech energy detection and wait for number of rings given in type.rings for “answer” (speech energy or ringing stopped), or call progress tone other than ringing, or “no answer.”

  This instruction handles differences between telephony types better than **tic(‘O’)**, or the combination of **tic(‘o’)** and **tic(‘W’)**.

- *D* — Dial *ctype.dialstr*; wait for any call progress tone, then resume the script.

- *F* — Flash; wait for any call progress tone, then resume the script.

- *O* — Originate (go off-hook and dial ctype.dialstr); wait for the first call progress tone (CPT), then resume the script. Note that without Full CCA, the first CPT could be a Ringback, with no indication of Answer or No answer disposition.

- *W* — Turn on speech energy detection and wait for number of rings given in type.rings for “answer” (speech energy or ringing stopped) or “no answer.”

- *a* — Answer the line (go off-hook).

- *d* — Dial *ctype.dialstr*, then resume the script.

- *f* — Flash the hook (transfer to another line), then resume the script.
- *h* — Hang up the line (go on-hook).
- *o* — Originate (go off-hook and dial ctype.dialstr), then resume the script.
- *w* — Wait for the number of rings given in type.rings for “answer” (ringing stopped) or “no answer”

Note: If a dial tone is expected during a call, it is recommended that you use options ’D’, ’F’, and ’O’ (or ’C’) instead of ’d’, ’f’, and ’o’. This prevents the dial tone from being interpreted as a hangup signal.

Sample Script Using Network Interface Instructions

In this network interface instruction example, the script directs the **tic** to use the flash hook function to transfer to another line. The script sleeps for 2 seconds. The **tic** then dials out on the current channel using the phone number stored in ch.PHONE_NUM. The script sleeps for 3 seconds, then quits.

```
DIAL_OUT:
tic(’F’)
jmp(r.0!= ’D’ End) /*No dial tone, error */
sleep(2)
tic(’D’,ch.PHONE_NUM)
sleep(3)
End:
quit()
```

Feature-Related Instructions

The **tic** instruction is also used by optional features available with the UCS 1000 R4.2. This includes primary rate interface (PRI), which can provide ISDN cause values, and Full Call Classification Analysis (CCA), which can provide special information tones (SITs).

For more information about the **tic** instruction, see Appendix B, Summary of TAS Script Instructions.

See the **setcca** instruction in Appendix B, Summary of TAS Script Instructions, when using Full CCA.

PRI Script Instructions

On incoming calls, the ANI, DNIS, redirecting number, and service type are provided by the optional ISDN PRI feature (when available from the switch). On outgoing calls, the called number, service type, bearer capability, and outbound ANI can be specified using the optional ISDN PRI feature.

Note: You can use the ANI and DNIS to specify an application for a caller by assigning *DNIS_SVC to a channel in the Assign Channel Service screen. Then assign ranges of ANIs and DNISs in the Assign Number Service screen. Specify the application in the **Service Name** field. See Chapter 3, “Voice System Administration,” in *UCS 1000 R4.2 Administration*, 585-313-507.

See Chapter 1, “Digital Telephony Interfaces,” in *UCS 1000 R4.2 Communication Development*, 585-313-213, for information on establishing the PRI interface. See *UCS 1000 R4.2 Administration*, 585-313-507, for information on administering and assigning PRI. If you have the Advanced

PRI Feature Package (available to selected business partners), see the *Intuity™ CONVERSANT® Advanced PRI Developer's Guide* provided with the feature package for more information about special ISDN PRI signaling needs.

The following are instructions used by the ISDN PRI feature:

- **tic**

  The supported **tic** instruction options are listed in Table 12. These options are used in the same manner for PRI as for T1 (E&M).

Table 12. tic Options Supported for PRI

Option	Function
a	Answer an incoming call
h	Disconnect (hangup) a call
o	Originate a call, but do not wait for disposition
C, O	Originate a call and wait for answer supervision
d	Dial touch-tone digits.

Some options to the **tic** instruction are not applicable to the PRI. These options are listed in Table 13 on page 104.

Table 13. tic Options Not Applicable to PRI

Option	Function
f or F	Switch hook flash
w or W	Wait for speech detection
D	Dial digits and wait for tones

The PRI implementation of the **tic('C')** (Call) or the similar **tic('O')** (Originate) instruction provides additional return code information beyond the T1 (E&M) and Tip/Ring interface implementations. *r.1* returns the ISDN cause value (if available) in the event of an incomplete call. These cause values are returned by the network and are passed through to the script. The cause value is also passed in register *r.1* upon a disconnect event. Table 33 on page 560 in Appendix B, "Summary of TAS Script Instructions", contains a list of ISDN cause values returned in register *r.1*.

The include (header) file (**/att/include/tas_defs.h**) provides macro definitions of these values. This file can be used by your application by including the following line in your script source file:

```
#include "tas_defs.h"
```

- The **setattr** instruction can be used to request the information provided from the network before starting the script. If the switch allows you to select the type of information provided for incoming calls, either calling

party number (CPN) — also called station identification (SID) — or automatic number identification (ANI) — also called billing number (BN) — can be requested:

- SID preferred – Request SID; if not available, request ANI
- SID only – Request SID only
- ANI preferred – Request ANI; if not available, request SID
- ANI only – Request ANI only

Note: The **setattr** instruction describes the environment in which scripts run. The **setattr** directives take effect before the script starts. Therefore, it is not possible to dynamically alter a script's attributes. For this reason, the **setattr** instruction should be used only once in each script. For example, the following script fragment always requests ANI, regardless of the DNIS.

```
Begin:
     strcmp (dnis.0,”6145555121”)
       jmp (r.0 == 0, gotdnis)
       setattr (ATTR_ANI)
     gotdnis:
```

The values in Table 14 on page 106 (defined in **tas_defs.h**) can be used with the **setattr** instruction to specify the type of calling party information being requested.

Table 14. tas_defs.h define values

ATTR_ANI	ANI only
ATTR_ANI_O	ANI only
ATTR_ANI_P	ANI preferred
ATTR_SID_O	SID only
ATTR_SID_P	SID preferred

For example, to request only the SID be returned in the CPN, use the following instruction:

```
setattr (ATTR_SID_O)
```

The following is an example of a script that uses **setattr** to request ANI.

```
#include “tas_defs.h” /* System provided header file */
#define NUMBER_TO_BE_CHECKED 0 /* User space allocation */
/* Specify attribute that causes the PRI to request ANI */
     setattr(ATTR_ANI)
/* Specify the speech file you wish to use */
     tfile(application)
Begin:
     /* Retrieve the ANI number */
     /* ANI is a character string stored in special register
 * ‘IE_ANI’ */
     strcpy (ch.NUMBER_TO_BE_CHECKED, IE.IE_ANI)
```

```
  /* Is the call from AREA code 614 ? */
  jmp (ch.NUMBER_TO_BE_CHECKED != '6', WrongAreaCode )
  jmp (ch.NUMBER_TO_BE_CHECKED+1 != '1', WrongAreaCode )
  jmp (ch.NUMBER_TO_BE_CHECKED+2 != '4', WrongAreaCode )

    /* Area Code OK */
    /* Answer call */
  tic('a')

  /* Begin transactions */
  talk("Welcome ... ")
  ...

  ...
  quit()

WrongAreaCode:
  /* Wait for caller to Hangup */
  goto WrongAreaCode
```

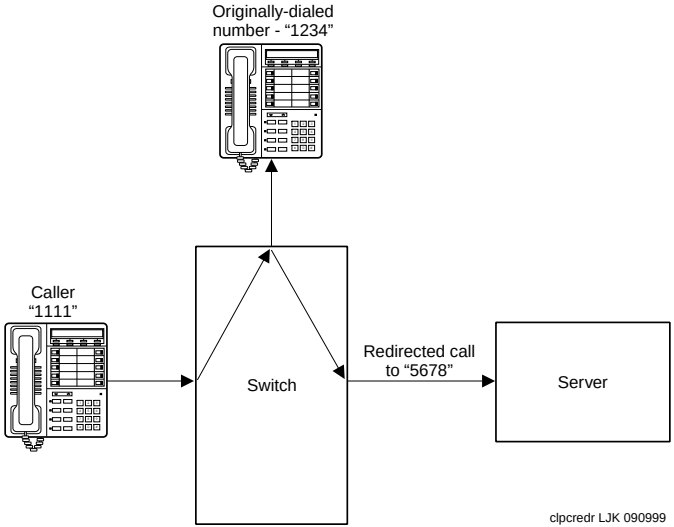
If an incoming call has been redirected from the originally-dialed number to a PRI trunk on the UCS 1000 R4.2, a redirecting number is available as an ASCII character string in the script register IE.IE.REDIRECTING. The redirecting number is the originally-dialed number.

Figure 7 on page 109 depicts a call placed to "1234" that has been redirected to "5678." In this example, the system provides the telephone numbers shown in Table 15 on page 108 to the script.

Table 15. Telephone Numbers Provided to the Script for Redirected Numbers

Script Register	Number
IE. IE_DNIS	“5678”
IE.IE_REDIRECTING	“1234”
IE.IE_ANI	“1111”

Figure 7. Retrieving the Redirection Number - Example



If the PRI trunks are provided with multiple incoming services (for example, MEGACOM800 or MULTIQUEST), the script provides a register that contains the type of service which delivered the incoming call. The service type is determined by the network processing the incoming call and is stored in the IE_SERVICE register as an integer. Service types are listed in the **setparam** instruction in Appendix B, Summary of TAS Script Instructions.

- **setstring (OUTBOUND_ANI, *ctype.src*)**

  The **setstring** instruction allows an application to set a CPN for an outbound call.

  The *ctype.dst* argument is a character string which represents the CPN. After **setstring** is invoked, subsequent outbound calls will use the *ctype.dst* argument as the outbound CPN.

  For example, the following instructions will place an outbound call to ch.CALLED as the dialed number and with (614)555-1212 as the calling party number.

```
setstring (OUTBOUND_ANI, ”6145551212”)
tic (‘o’,ch.CALLED)
```

Note: If the **setstring** command fails, *r.0* is set to a negative number. Note that the **setstring** command failed if the destination operand (OUTBOUND_ANI) is incorrect or if the format of the number to use for outbound ANI is incorrect.

- **setparam (*type.param, type.value*)**

  The **setparam** instruction sets a parameter associated with a script. For example, **setparam** can be used to change the service type and bearer capability for outbound PRI calls by setting the SERVICE_TYPE and BEARER_CAP parameters. Valid parameters and their values are listed in Appendix B, Summary of TAS Script Instructions.

Miscellaneous Instructions

The following are miscellaneous instructions used in TSM scripts.

- **chantype ()**

  The **chantype** instruction is used while implementing Converse Data Return to support the DEFINITY Call Vectoring feature. The instruction enables scripts to determine the type of channel they are running on.

  Table 22 on page 430 in Appendix B, Summary of TAS Script Instructions, contains a list of values returned in register *r.0* from the **/att/include/irDefines.h** header file. A negative value is returned if an error occurs.

For example:

```
#include “/att/include/irDefines.h”

/* get channel type */
chantype()
load(int.F_chantype, r.0)


/* channel type must be TR or LSE1/LST1 */
jmp(int.F_chantype == IRD_TR, L__chan_OK)
jmp(int.F_chantype == IRD_ASAI, L__chan_OK)
jmp(int.F_chantype == IRD_LST1_DEF, L__chan_OK)
jmp(int.F_chantype == IRD_LST1_GAL, L__chan_OK)
jmp(int.F_chantype == IRD_LST1_ASAI, L__chan_OK)
```

- **hbridge*(type.src,type.src)***

  The **hbridge** script instruction directs the current channel to bridge partially to another channel. This results in the audio coming in on the specified channel to be heard or dropped by the calling party (current channel). The specified channel does not hear the calling party. The current channel does not hear voice responses or other background audio on the specified channel.

The first *type.src* argument is a valid channel number. The second *type.src* argument is either 1 to add the specified channel or 0 (zero) to drop the channel. Values for the channel numbers and the add/drop flag follow the conventions for all *type.src* arguments (see Script Conventions on page 25).

If the **hbridge** instruction is not successful, a negative value is returned to r.0. The following are conditions under which the **hbridge** instruction may fail:

- A **hbridge** attempt to a current channel failed.
- The channel reached its limit for listen time slots (maximum seven per channel).
- A system call failure occurred.

- **hundsec(*type.dst*)**

  The **hundsec()** instruction loads the integer *type.dst* with the system time in hundredths of a second.

Note: Do not use the **hundsec** instruction in a loop to insert delays in script execution. Use the **sleep** or **nap** instructions instead.

- **listenall(*type.src*, *type.src*)**

  The **listenall** instruction listens to all audio input on a specified channel. Audio input includes normal voice responses to the network. The specified channel does not hear any audio from the current channel. This allows administrators to monitor the channel.

The script with the call to **listenall** must be kept running until the caller is finished monitoring the audio input on the other channel. One way to accomplish this would be to add a call to sleep directly after **listenall** instruction. For example:

```
listenall (45, ADD)
sleep (45)
```

These instructions keep the monitor script running for 45 seconds after the script starts. You must determine how long the other channel will be monitored and use the appropriate sleep value.

The first *type.src* argument is a valid channel number. The second *type.src* argument is either 1 to add the channel or 0 (zero) to drop it. These arguments must follow the conventions for *type.src* arguments discussed earlier in this chapter (see Script Conventions on page 25).

If the **listenall** script instruction is successful, a positive value is returned to r.0. If the **listenall** instruction is not successful, a negative value is returned to r.0.

The following are reasons the **listenall** instruction might fail:

- An attempt to monitor current channel failed.
- An attempt to monitor more than one channel failed.
- The channel reached its limit for listen time-slots (maximum seven per channel).
- A system call failure occurred.

Note: If the **listenall** instruction hears a dialtone, the instruction may disconnect depending on how the script uses the event instruction. See the **event** instruction in this chapter or see Appendix B, Summary of TAS Script Instructions.

- **trace(*type.src* [,*type.src*])**

  The **trace** script instruction works with the **/vs/bin/trace** UNIX line command to monitor the progress of scripts. This capability is useful in debugging and troubleshooting scripts, either during initial application development or if problems rise while the application is running. The **trace** instruction enables TSM to print messages to the shared memory area for trace messages. These messages can include the default trace messages for TSM or a specific channel.

Note: If there are too many traces running simultaneously on a system, the buffer in which this information is stored may be filled and some data lost, with no notice of this in the trace output.

The first argument is evaluated as a number and is used as a step identifier. The optional argument can be used to print a specific data value of interest. The optional argument may be any integer type, or a null terminated character string.

In the following example, 25001 and int.F_TEMP are traced.

```
trace (25001, int.F_TEMP)
```

When the example **trace** statement above is run, the statement appears as step 25001 in the trace and the content of F_TEMP is displayed.

See *UCS 1000 R4.2 Administration*, 585-313-507, for additional information on the **trace** command.

Script Development

The following information details defining header files and user memory, identifying events, and specifying source files.

Transaction Control Header Files

Many parameters used in a script are defined in header files. The first two are defined already; the third and fourth files are defined by the application developer. The user directories and application names shown are only examples.

1 Defines generic messages to the DIP and their format

/att/msgipc/tsm_dip.h

2 Defines speech codestyle messages to TSM

/att/include/codestyle.h

3 Defines script variables and allocates user memory

/usr/var/appl*N*/trans/*application_name*def.h
or
/att/trans/sb/*application_name/application_name*def.h

4 Defines the application messages to the DIP and their format

/usr/var/appl*N*/dip*N*/tsmdipappl.h
or
/att/trans/sb/*application_name/application_name*def.h

Defining User Memory

User memory is defined by the **mkheader** command (see *UCS 1000 R4.2 Administration,* 585-313-507, for more information). The **mkheader** command allocates space for local, global, and database variables used by the script. The program is initiated by entering:

mkheader *application_name*

This command creates a header file called ***application_name*def.h**.

Identification of Events

Once the information that is to be recorded during a transaction has been determined, then a number is assigned to each noteworthy event and a label is entered for that event.

When an event occurs during a transaction, the script can increment the event or load the appropriate value into it. When the transaction is complete, the contents of event memory are passed automatically to CDH, which puts this data in the call data and call summary tables in the database.

Events are recorded in three ways:

- A count event increments an integer into event memory
- A store event loads an integer or string into event memory
- A time event loads the time into event memory

The following are examples for recording information about **getinput**:

getinput(ch.YN, 1)	/* Request Yes/No response */
incr(ev.1, 1)	/* Record event 1 */
load(ev.2, ch.YN)	/* Record event 2 variable name */
load(ev.3, time.0)	/* Record event 3 */

After getting a yes or no reply and storing it in a field called YN, the program increments event 1, which represents the number of attempts to get a yes or no reply; saves the integer in event 2; and saves the time of the response in event 3.

Note that events are “write-only” variables. Current event values cannot be read; they must be saved in temporary variables to be examined.

Source File

The script instructions are initially stored as an **a*pplication_name*.t** source file. This file is given as the argument to the **tas** command to produce a machine readable ***application_name*.T** file. The ***application_name*.T** file is stored in the **/vs/trans** directory and is used by the TSM process.

For more information on the **tas** command, see *UCS 1000 R4.2 Administration*, 585-313-507.

Wait Conditions

The TSM program is responsible for running scripts that are produced with an editor and compiled with the TAS program. Once a script is started on a particular channel, TSM continues to execute instructions until a wait condition occurs. Script execution then is suspended on that channel until the wait condition is satisfied by an external event or a timeout occurs.

Wait conditions fall into two general categories.

- Speech-flushing instructions — Script instructions that cause a wait by flushing any speech that has been queued for playing by the script before the instructions are executed.
- Wait-causing instructions — Script instructions that cause a wait during their execution that is characteristic of their function. They make a request on behalf of the script that must be satisfied by a process external to TSM.

Note: Some instructions fall into both of the wait condition categories.

Speech-Flushing Instructions

Some instructions cause the script to wait by forcing any speech phrases or text that have been queued by the **talk, ftalk**, **tchar**, **tnum**, **fsay**, or **say** instructions to play before the instruction itself is executed. Thus, the actual wait occurs before the instruction is executed. These instructions are:

dbase()
exec()
execu()
getdig()
getinput()
nwitime()
phremove()

phreserve()
quit()
setftalk()
setalk()
sleep()
sr_talkoff()
subprog
talkresume()
tfile()
tflush()
tic()

If any of these instructions are executed while there is speech queued for playing, the speech is played and the script waits for the play to complete before executing the instruction. Playing speech also causes any touch tones that have been received by the script and not yet retrieved with **getinput** or **getdig** to be thrown away unless the **setttfl** instruction has been used to enable the type_ahead feature.

One exception, where a wait for speech is not caused, is when the **tflush** instruction is used with its second argument set to 1. This causes any queued speech to be played and *spun off*; the script continues execution without waiting for the play to complete.

There is no timeout imposed by TSM on a wait for speech to finish playing. TSM depends on a message from the voice system to tell it when to resume script execution after the play has stopped.

Wait-Causing Instructions

Some instructions make a request of a process external to TSM and cause the script to wait until that request is satisfied. In addition to causing a wait for a request to be satisfied, some of these instructions are also speech-flushing instructions (see above) and so may cause a wait for speech to finish playing before they are executed.

The wait-causing instructions are listed below. See the descriptions of these instructions earlier in this chapter for more information.

- The **dbase** instruction sends a message to a DIP and may wait for a return message. The **dbase** instruction does not cause a wait if it is used to send a message to a DIP and not to receive one in return. (In this case, the number of bytes expected for the return message is set to a negative integer.) The timeout period for **dbase** is 45 seconds by default. Change this timeout period with the **nwitime** instruction.
- If the **getinput** or **getdig** instruction is executed specifying a number of digits greater than that which has already been entered by the caller, the script waits for the required number of digits to be entered. Use the **ttdelim** instruction to set delineators that allow **getinput** or **getdig** to accept variable length digit strings without waiting for a timeout. Two timeouts affect the **getinput** or **getdig** instruction: an initial timeout and an interdigit timeout. Both of these timeouts are five seconds by default.

Change the default timeout with the **tttime** instruction. Interdigit timeout applies touch-tone input, Dial Pulse Recognition and Whole Word speech recognition.

- The **phreserve** instruction is used to reserve space in a talkfile for a phrase. The script waits for a message from the voice system to complete the request. The timeout period for **phreserve** is 45 seconds by default. Change the timeout value with the **nwitime** instruction.
- The **resource_alloc** instruction may cause a wait if the optional mode argument is used to wait for a resource if it is not immediately available.
- The **sleep** instruction causes the script to wait for the specified number of seconds before continuing. The **nap** instruction causes the script to wait for the specified number of centiseconds (hundredths of a second).
- The **sr_talkoff** instruction will wait until the barge-in capability is turned on or off for speech recognition before returning to the script.
- The **talkresume** instruction is used to play speech that is *remembered* for the channel (that is, played with the third flag of the **tflush** instruction set to 1). As with **tflush**, there is no timeout required for this instruction. The voice system informs TSM when the playing has completed.
- The **tic** instruction has several functions that constitute the interface between the script an the telephone network. Most **tic** functions cause a wait condition while the function is being completed. The timeout period

for **tic** is 45 seconds by default. Change this timeout value with the **nwitime** instruction or modify if implicitly by some **tic** functions (see **tic** for more details).

- The **tstop** instruction is used to stop all speech playing or coding activity on the channel. Depending on what argument is passed to **tstop**, the script may wait for a message indicating that such activity has stopped before continuing. The timeout period for **tstop** is 45 seconds by default. Change this timeout period with the **nwitime** instruction.
- The **vc** instruction is used to do voice coding (recording speech from the caller). Two timeouts affect the **vc** instruction: an initial silence timeout and an interword silence timeout. Both of these timeouts are five seconds by default. Change this timeout period with the **vctime** instruction.

Avoiding Common Pitfalls with Wait Conditions

Once TSM is executing the instructions of a script, that execution proceeds uninterrupted until a wait condition occurs. Normally, at this point, script execution is suspended until the system function which required the wait is completed, then the script resumes execution at the point where the wait occurred.

Note: Scripts that contain more than 400 instructions without a wait condition are suspended by TSM until either an event is received on that channel or the TSM event queue is empty. This may cause

the script to execute with noticeable delays under heavy system load. To avoid this, design your scripts to include wait conditions (at least one per every 400 instructions).

Several things can happen during a wait which may effect the script's execution after that point. When a script needs to wait, TSM returns to reading its message queue to process external events that effect the execution of all currently running scripts. The following is a list of some of the actions TSM may take:

- TSM may resume execution of a waiting script on another channel where the conditions of the wait have been satisfied or the wait has timed out.
- Instead of a wait condition being satisfied normally, TSM may receive another event for the channel, such as a caller disconnect, which will terminate the script.
- The MTC process may seize equipment being used by the channel causing the script to be terminated (if the seizure is done unconditionally).
- Touch tones typed by the caller are received by TSM and copied into the script's touch-tone buffer during a wait.

- Events that are handled by the **event** script instruction may cause the current wait to be interrupted and script execution to be resumed with an interrupt subroutine (see the event instruction for further details). When this is done, the script may return to the point of interruption (and resume the wait) if the interrupt routine has not caused a second wait condition. If the interrupt routine does cause a wait, the original wait condition will be disregarded and the script will continue at the next instruction after the point of interruption when the routine returns (using the **rts** instruction).

It is important to remember how timeout values apply to wait conditions. The **nwitime** instruction may be used to change the general next wait instruction timeout (NWIT), which has a default of 45 seconds. This timeout value only applies to the next **background**, **dbase**, **phreserve**, **sr_talkoff**, **tic**, or **tstop** instruction wait. It does not affect the timeouts of other wait-causing instructions that have their own specific timeout values (see Wait-Causing Instructions on page 122). Nor does it affect the wait for speech to finish playing, which has no practical timeout. The NWIT is reset to the 45 second default when the second instruction after **nwitime** is executed.

Do not to let a wait condition separate a decision point in a script and its dependent action point if the decision is affected by what may happen in the system during the wait. An example of this is using the **scrinst** instruction to take action based on the number of instances of a particular script running at a particular time. The **scrinst** instruction returns the number of instances of a script at the time it is executed. If a wait condition is allowed between the **scrinst** and the point in the script where action is taken based on the result of

scrinst, an unintended consequence may result because the number of scripts running may have changed during the wait. In this case, use **tflush** before **scrinst** to make sure that any wait for speech playing will not be done at a critical time and take care not to use any other wait-causing instructions in the critical interval. This is especially important when using the **exec** instruction based on the result of **scrinst** since **exec** is a speech-flushing instruction. (See scrinst([ctype.script]) on page 79 for an explanation.)

Troubleshooting Scripts

This section contains information to help script writers make sure their scripts are working properly. Included are procedures for detecting problems in a script, possible problems found in scripts, and points to keep in mind when using specific instructions. The following points are discussed:

- Checking the status of **talk** instructions
- Erasing arguments in the **ttdelim** instruction
- Checking for string matching failures
- Losing touch tones

See *UCS 1000 R4.2 Administration*, 585-313-507, for more information on the **trace** command.

Check the Status of ftalk or talk Instructions

One or more **ftalk** or **talk** instructions in a script cause a list of phrases to be played. If a failure occurs in the process of playing one or more phrases, the UCS 1000 R4.2 software plays as many phrases as possible but returns an error code of -1 in r.0. When using the **tflush** instruction, the script writer can tell the script to check r.0 for the returned status.

When **ftalk** or **talk** instructions are executed, the system queues phrases in a buffer, but the phrases are not immediately played. Phrases are played when:

- The script executes any wait-causing script instruction (the most common occurrence)
- The **tflush** instruction flushes the buffer

In order to check the status of a list of phrases being queued, the script must examine r.0 immediately after one or more talk instructions have been executed. By entering the **tflush** instruction in the script, the status can be obtained immediately following one or more **ftalk** or **talk** instructions. The **tflush** instruction, like other speech-flushing instructions, causes the phrases queued by the **ftalk** or **talk** instructions to be played. But only **tflush** retains the returned status from playing phrases in *r.0*. The other instructions may overwrite *r.0* with their own return status.

The following is an example of using *tflush* to examine the returned status of queued phrases:

```
talk (“Hello”)
talk (200)
talk (“Enter your ID”)
tflush()
jmp(r.0 < 0,play_fail)
getinput(ch.ID, 4)
--
--
--
play_fail:
quit(3)
```

In this example, several phrases are queued as the result of the three talk instructions. The **tflush** instruction is optional and does not have to be used. Its function is to allow the script writer to check the status when phrases are played. When the **tflush** instruction is completed, r.0 contains 0 if no errors occurred; otherwise, it contains -1. If no errors occur, the script collects touch tones by executing the **getdig** instruction. If there are errors in the play, control jumps to the label play_fail.

It is recommended that the **tflush** instruction be used only after several **ftalk** or **talk** instructions have executed, or after every **ftalk** or **talk** instruction. Each time **tflush** is executed, two interprocess messages are sent: TSM sends a message to a system process which causes the phrases to be played and the system process returns a message to TSM which contains the status. These processes can delay play of a script if **tflush** is used too often.

Note: Take care to avoid program loops that queue up large numbers of phrases before playing them with **tflush** or other speech flushing instruction. TSM uses dynamic memory allocation to store the phrase queue. Allowing the queue to become too large before flushing it could cause the TSM process to become too large, negatively affecting system performance (60 phrases is a reasonable queue limit).

Erase Arguments in the ttdelim Instruction

The following points must be kept in mind when using erase-character and/or erase-entry arguments in the **ttdelim** instruction.

1. If a **getinput** or **getdig** request asks for *x* touch tones and *x* are entered, neither this string nor the last character of this string can be erased using an erase-entry or erase-character, respectively. Once an input request for a specified number of touch tones has been satisfied, it is too late to

perform an erase function. Hence erasing a string or the last character entered must be done before the last touch tone satisfying the input request has been entered.

For example, if a 5-digit string is requested, and a caller has entered 8275, it is possible at this point to erase the 5 in the string 8275. However, once another digit is entered, the result is immediately processed by the script. One exception arises and is explained as number 2 below.

2 If two touch tones are used as an erase character and/or erase-entry argument in the **ttdelim** instruction, the first touch tone of the argument should not be one that can be part of a *normal* input string.

For example, suppose a script will accept a 5-digit ID as input. Normal input in this case consists of any 5-digit touch-tone string comprised solely of digits. Suppose that the following **ttdelim** instruction appears in the script:

```
ttdelim ('#','6#',-1,-1)
```

Here, # is used to erase the last character and 6# is used to erase the last string entered.

Whenever the first character of a two-character erase argument is entered, the script always waits for another touch tone to be entered to determine if it is the second character of the two-character erase argument.

A problem arises with the preceding use of the **ttdelim** instruction. Assume that a caller enters 28136 in response to a request for five touch tones. The script will not immediately process these five touch tones. The system waits for a # to be entered because the 6 is the first of a 2-character erase argument. If the caller does not enter any more touch tones, the request for five touch tones times out. In this example, it is impossible to enter IDs that end with the digit 6.

To avoid this problem, use either single character erase arguments or, if 2-character erase arguments must be used, make sure that the first character cannot be part of a normal input string. The problem in this example can be solved by simply reversing the 6 and #. The new **ttdelim** instruction would be:

```
ttdelim ('#','#6',-1,-1)
```

In this case, 28136 would be immediately processed by the script.

Speech String Matching Failures

Occasional speech string matching failures can occur when a *substituted* or abbreviated string in a **talk** instruction is searched in a listfile when **tas** assembles a script. If the string fails to match, add one or more words to the string in the **talk** instruction until the string matches.

Although the string matching algorithm has this small drawback, it is user-friendly in that it requires a minimum amount of effort on the part of the script writer to identify phrases in **talk** instructions. Rather than entire phrases in **talk** instructions, only a minimal substring that uniquely identifies the phrase in **talk** instructions is required.

Loss of Touch Tones

Script instructions can be grouped in two categories:

- Those that cause the touch-tone and phrase buffers to be flushed as a preliminary step before the instruction is executed. These instructions are the speech-flushing instructions listed earlier.
- Those that do not flush the touch-tone and phrase buffers

You also should be aware of **setttfl** and **ttclear** instructions that help control touch-tone loss.

The decision to have certain instructions (those listed in the first category) flush the touch-tone and phrase buffers is based on the need to keep the caller and script *in sync*. Without flushing these buffers at certain points in the script (for example, after the speech-flushing instructions are executed), the caller and script may get so far *out of sync* that the caller gets confused and must hangup and call again.

To illustrate this situation, consider the following example where touch tones are not flushed periodically. A script prompts a caller to enter the following:

- A 4-digit id
- A 2-digit code to select from services described by the script
- A 1-digit code for additional information on a specific service

In response, the caller enters the following touch-tone sequence:

8225
31
6

These touch-tone sequences identify customer 8225 requesting service 31 and entering 6 to obtain various prices of this service. When callers become familiar with a script, they often enter touch tones before the script prompts for them. If all touch tones are retained and not flushed periodically, it is possible for the script to understand all the touch tones entered ahead of the script. However, when callers enter incorrect touch-tone sequences, it is difficult for both the script and the caller to take immediate corrective action. For example, suppose a caller enters the following touch-tone sequences before the script asks for them:

8225
37
6
14

2
88
5

If the entry 37 identifies a valid service but the caller meant to enter 31, then it is difficult for the script to recover from an error. If all touch tones are retained, they are processed and the caller cannot stop the processing. Moreover, the caller may not realize there is an error and be confused by what the script plays back. The script and the caller become further *out of sync*.

Situations like the one just described can be prevented by clearing the touch tone buffer periodically. Experience has shown that this is generally a more user-friendly approach. Although touch tones are occasionally lost if users enter them too far ahead, the only penalty is that users must re-enter a single response. However, the main advantage of periodically flushing the touch-tone buffer is that it makes writing scripts simpler. If all touch tones are retained, the variety of error-recovery situations that occur is large and nontrivial if it must be done in the script. If the touch-tone buffers are flushed, the script writer is relieved of the addressing error-recovery situations in the script. The **setttfl** instruction can be used to prevent touch tones from being flushed. See the description of the **setttfl** instruction earlier in this chapter.

4 Data Interface Processes

Overview

This chapter describes the standard interface between a data interface process (DIP) and the transaction state machine (TSM) scripts and the logger/alerter. It also details the pieces involved in writing DIPs, including the C-library functions and transaction assembler script (TAS) instructions.

This chapter assumes the following:

- The C-development software is installed.
- You are familiar with C-language programming in a UnixWare operating system environment.

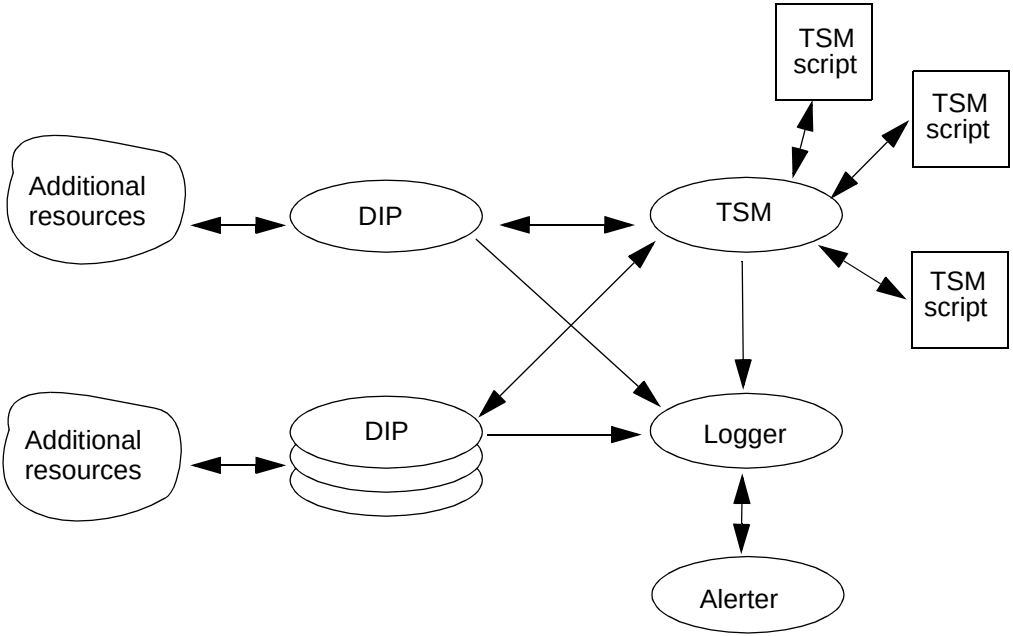
Note: All new DIPs should be written in terms of the IRAPI. See Chapter 5, IRAPI, for additional information.

Introduction to the Data Interface Process

In any application, TSM scripts control how a call is handled. Decisions and actions such as answering the phone or collecting touch-tone digits are specified using Script Builder or the TAS assembly-like language. However, TSM scripts alone cannot handle a significant number of applications that need to access external data from files or a database or perform complex numerical calculations. A DIP provides these capabilities. In fact, DIPs provide all the resources of C-language programs and the UnixWare operating system to scripts.

A process is a program that is currently running in the system. TSM, logger, alerter, and DIPs are examples of processes. Figure 8 on page 138 shows typical interaction between a DIP and other processes in the voice system software.

Figure 8. Data Interface Process (DIP) Architecture



Generally, a DIP interacts with the following processes:

- TSM scripts
- Additional resources (for example, a database or host computer)
- The logger

DIPs are usually message-driven, meaning they wait until a message arrives before taking any action. Once a request is received from a TSM script, for example, the DIP processes the message and returns the results to the corresponding TSM script.

The overlapping circles in Figure 8 on page 138 indicate that a DIP can have multiple copies of itself running and reading from the same message queue to allow for faster servicing of requests.

Message Queues

DIPs talk to TSM scripts through UnixWare system interprocess communication (IPC) messages queues. IPC message queues are similar to mailboxes behind the registration desk in a hotel.

The voice system acts as the hotel, the guests correspond to the DIPs, and the attendant at the desk is TSM. The DIPs have their own separate mailboxes (messages queues) for receiving messages sent by other guests or outsiders.

Data is passed between DIPs and TSM scripts through these mailboxes. DIPs leave messages to TSM scripts in TSM's predefined mailbox. TSM then reads, sorts, and distributes these messages to the appropriate script, just as the attendant distributes messages left for guests at the front desk of the hotel.

The voice system has a total number of 95 message queues (mailboxes) available, numbered from 1 through 95. These numbers, known as message queue keys (Qkeys), serve to uniquely identify individual message queues.

The voice system Qkeys are divided into the following groups:

- Voice system processes: 1–19
- Hardcoded DIPs: 20–54
- Other processes: 55–63
- Dynamic processes (including DIPs): 64–95

Hardcoded and dynamic DIPs are discussed in the Types of DIPs on page 141 section of this chapter.

For additional information about UnixWare System Message Queues, see the *UnixWare Programming with System Calls and Libraries.* For additional information about UnixWare programming features, see the *UnixWare Programming in Standard C and C++.*

Types of DIPs

There are two types of DIPs: hardcoded and dynamic. Both may exist on your system. The only difference between these two types of DIPs is the manner in which they are assigned their message queue number.

A hardcoded DIP has a pre-defined message queue, or DIP number, in its C-code. Using the hotel example, it is similar to selecting a mailbox without first checking at the registration desk to make sure no other guest is using that mailbox. DIPs reading from the same message queue will interfere with each other.

Dynamic DIPs (DynaDIPs) avoid this type of conflict by *asking* the voice system for an available message queue at run-time. That is, DynaDIPs do not know what message queue they will have until each time they are run on the voice system. Using the hotel example, it is similar to asking for an available mailbox for a guest, rather than asking for a mailbox number.

Each DynaDIP gives its DIP name and instance number to the system and a unique, unused Qkey is returned. DIPs using the same name receive the same Qkey from which to read, allowing for DIPs that are instances of each other. In this case, the DIPs intentionally read from the same Qkey because they are instances of one another. However, you should avoid having two unrelated DIPs use the same name and then read from the same Qkey. Using unique names instead of numbers (Qkeys 1–95) reduces the chances of clashes between two unrelated DIPs.

Note: It is *strongly* recommended that you use DynaDIPs instead of hardcoded DIPs.

Message queue assignments remain in effect and are fixed as long as the voice system is running, despite DynaDIPs dying or respawning. Restarting the voice system removes these assignments, as the name “Dynamic” DIPs stresses the fact that their message queues are dynamically assigned and likely will change across restarts of the voice system.

The voice system allows up to 32 dynamic and 35 hardcoded DIPs. However, the following caveats apply:

1. The type of work DIPs and other processes do affect the performance of the system. Thus, the actual number of DIPs that can run with acceptable performance might be less than 32 dynamic and 35 hardcoded DIPs.
2. The voice system tunes the UnixWare system for a maximum of 75 processes running at one time. You might need to increase this tunable parameter to fit all your DIPs and all the other processes in your specific system.

Bulletin Board

The bulletin board (BB) is an area of memory used for registering voice system processes and DIPs. Expanding again on the analogy of the voice system as a hotel for DIPs and other processes, the BB is like the registration book of this hotel. Before interfacing with any other process, DIPs start and

register themselves by DIP name, instance, number, and assigned Qkey in the BB. Each DIP instance is assigned one of the fixed number of available slots. There are 111 slots: slots 1–79 are reserved for hardcoded processes and slots 80–111 are for dynamic processes. Slots cannot be shared even if the DIPs share the same message queue.

DynaDIPs must register in the BB, as they can only receive a dynamically assigned message queue after *checking-in* at the front desk.

When the voice system starts, there is a typical influx of DIPs trying to register themselves at the same time. Registration is done in an orderly first-come first-served basis, as in a well-run hotel.

Besides getting an assigned, unused Qkey, there are two other advantages to posting a process in the BB:

- A process is protected from having duplicate copies of itself running. That is, only one process is allowed to run with a specified name and instance.
- Based on the rules supplied in the **/vs/etc/iCk.rules** file, the integrity checking (iCk) process checks all processes specified by the rules file to determine if they are stuck or not. Being *stuck* means that they have started processing a message but have not completed the process in the period of time specified by the rules file. If a process is stuck, iCk responds in one of three ways:
  - ~ Reports the process to the logging system

- Reports the process and kills it
- Reports the process and then executes a specified command to correct the issue

BB Slots

It is possible that, in time, the BB may be filled with posted processes, preventing your process from being posted. Use the **bbs** command to display the contents of the BB and to determine if it is full. Remember that slots 80–111 are for DynaDIPs. If the BB slots are full, stop and restart the system. This usually frees some slots so that you can post your process. See the *UCS 1000 R4.2 Administration,* 585-313-507, for additional information on the **bbs** command.

Writing the DIP

A DIP must be able to send and receive messages to and from TSM. A DIP may send any errors or other information to the logger. DynaDIPs send and/or receive messages using the same method, format, and library functions as hardcoded DIPs. Messages are sent to and received from the appropriate Qkey by specifying the corresponding Qkey. The Qkey must be known before any message can be sent or received.

Designing a DIP includes the following steps:

1. Define the data to be passed between the DIP and the TSM script
2. Initialize the DIP to the system
3. Send and receive messages
4. Implement the application-specific processing
5. Define and add logger errors (optional but recommended)
6. Add error reporting (optional but recommended)
7. Add trace messages (optional but recommended)
8. Compile and execute the DIP.

The examples provided here are used in the template for writing a DynaDIP. See the Sample DIP on page 393 in Appendix A, Application Example.

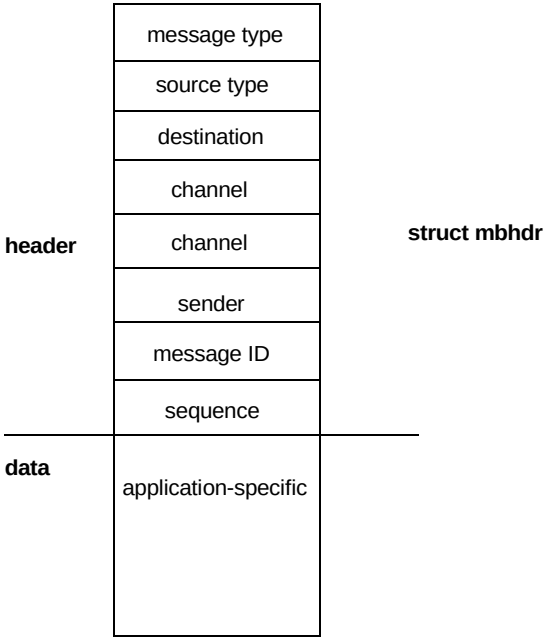
Step 1: Define Data to be Passed Between the DIP and the TSM Script

Before writing the actual DIP, you must first define the data that is to be passed between the DIP and the TSM script. First, the data to be sent is packaged or formatted as a message just like a letter is enclosed in an envelope. Once sent, the recipient gets the data by unpackaging the message, or opening the envelope.

Message Format

Messages are defined generally in two parts: the header and the application-specific data. Both of these parts are specified in C-language structure. The header contains information about the addressee or sender, the voice channel number associated with the message, and the message id, as shown in Figure 9 on page 147.

Figure 9. Voice System Message Components



Header Components

In the message header file (**/att/include/mesg.h**), the voice system defines the header structure of IPC messages for DIPs and voice system processes. Figure 10 displays the header structure for these IPC messages.

Figure 10. Message Header Structure

```
struct mbhdr {
    long mtype;   /* Message type */
    short    irType;  /* Source type for message */
    ir_key_t irWhoto;/* Destination queue or channel owner */
    long     irChan;  /* Channel number */
    long     mchan;  /* Channel number */
    short    morig;   /* Sender's Qkey */
    short    mcont;  /* Message id */
    unsigned short mseqno; /* Message sequence number */
};
```

Note: As you read the following field descriptions, see the Sample DIP on page 393 provided in Appendix A, Application Example. The fields in the header structure are as follows:

- The *mtype* field allows more control over the destination of messages. This field is used only when sending messages from one DIP to another.

Note: The *mtype* field is not often used, but must be a positive non-zero number. Set this field to 1 (one) if you do not plan to use it.

- The *irType* field indicates the source type for message. This value is set by the function used to send the message. This field can be ignored if the DIP is written not using IRAPI.
- The *irWhoTo* field specifies the Qkey for the destination queue or allows the IRAPI to specify that the message be sent to the channel owner. This field can be ignored if the DIP is written not using IRAPI.
- The *irChan* field specifies the channel number. This must be a valid channel number when irWhoTo == IRAPI; otherwise it should be set to IRD_INVALID. This field can be ignored if the DIP is written not using IRAPI.
- The *mchan* field refers to the channel number that determines which TSM script is to receive the message. Messages sent from a DIP to TSM are routed to the TSM script running on the specified channel. This field originally is set by TSM (discussed later in this section) and *must* be returned to TSM.
- The *morig* field specifies the Qkey of the sending or originating process. A DIP's Qkey is returned by **VSstartup** for DynaDIPs or **irRegister** for IRAPI processes, or is defined for a hardcoded DIP from the list in **mesg.h**. This field *must* be used for returning a message to the sending process.
- The *mcont* field (also referred to as the message id) specifies what type of data is contained in the message from TSM to the DIP, which allows the handling of messages with data of all shapes, sizes, and meanings.

Message ids are usually defined in the DIP **.h** file (<*dip_name*>**.h**), and can be used in the TAS **dbase** instruction. Without this message id, the DIP is unaware of the kind of data received. It is important to use this field if a DIP can process more than one type of data input.

For an application to identify each different message to be sent or received, select a unique positive number. In the mesgrcv on page 600 example in Appendix C, C-Library Functions, two different messages are defined. One is CALLER_INFO that is set to 6910 and the other is ORDER_AMOUNT that is set to 6930.

- The *mseqno* field allows more control over the sequencing of messages. This field is not used often.

Data Components

The application-specific data part of the message follows the header. The application is free to shape and size the data in the way it chooses within system-imposed limits. For every different type of data received, there may be a corresponding structure and unique message id. A complete set of messages to be received can be represented in C-language by a union of message structures as shown in this example from a stock application (Figure 11 on page 151).

Figure 11. Message Structure Union Example

```
/* Define all message structures that can be received. */

struct stockInfo {
   struct mbhdr  hd;
   int   stockId;
};

struct callerInfo {
   struct mbhdr  hd;
   charcallerName [30];
   int callerId;
};

/* Define the union of all possible message structures that can
be received.*/

union rcvMsg {
   struct ms_univ u;/* standard message */
   struct stockInfo stock;
   struct callerInfo caller;
};
```

Figure 11 on page 151 shows the union of received message (**rcvMsg**). Note that this message structure is as large as the largest message expected in the application, thus it can be used to hold any message read. Similarly, the set of messages to be sent can be defined in another union.

The union example contains the message structure **ms_univ**, defined in **mesg.h**, that consists of four long integers as shown in Figure 12.

Figure 12. Standard Message Structure

```
/* universal structure for passing a message */

struct ms_univ {
    struct mbhdr   hd;
    long   arg[4];
};
```

At this point, you should have defined the data to be passed between the DIP and the TSM script. Proceed to Step 2: Initialize the DIP to the System on page 152 to add the C-code that initializes your DIP to the voice system.

Step 2: Initialize the DIP to the System

When starting up, a DIP should do the following:

1. Identify itself to the voice system by posting itself in the BB
2. Set up the tracing facility
3. Get its assigned message queue, if it is a DynaDIP

Two C-library functions (**VSstartup** and **startup**) are available to perform the above activities. These two functions are identical, but **VSstartup** is used for DynaDIPs and **startup** for hardcoded DIPs.

DynaDIPs

The following functions are used to initialize DynaDIPs.

VSstartup

VSstartup is called once to post a process, like a DIP, to the BB. It also sets up the trace facility. **VSstartup** takes the DIP name, its instance, and a DIP flag. DIP flag can take one of two values, constants DIP_PROC or NONDIP_PROC. Setting the flag to the constant DIP_PROC allows the DIP to send and receive messages to and from TSM scripts. If the flag is set to the constant NONDIP_PROC, messages sent by the DIP to TSM scripts are ignored by TSM. An assigned IPC message Qkey is returned if successful as in Figure 13. A negative value is returned if an error occurs.

Figure 13. VSstartup Input and Output

The DIP name should be a unique printable name of up to 15 characters.

Note: Any application dependent initialization, such as opening files, should be included.

Figure 14 displays the **VSstartup** synopsis in C-code for the dynamic DIP.

Figure 14. VSstartup Synopsis

```
#include <sys/types.h>
#include "VS.h"

key_t VSstartup(dipName,instance,flag)
char   *dipName;/* unique name associated with process */
short  instance;/* process instance */
long   flag;/* Will DIP talk to TSM scripts? */
```

Note: Normally, a system that has a significant number of channels will take time to reach the inserv state for all channels due to the diagnostics that are run on the channels at startup. If this is the case with your system and the DIP depends on all channels being in service, you may consider delaying the initialization of the DIP by adding a **sleep** instruction prior to **VSstartup** or other initialization processes.

For additional information, see VSstartup on page 615 in Appendix C, C-Library Functions.

VStoqkey and VStoname

After posting themselves in the BB using **VSstartup**, DynaDIPs must retrieve the Qkeys of all other user-defined DIPs to which they send or receive messages, as shown in VStoqkey on page 619 in Appendix C, C-Library Functions. The function **VStoqkey** converts DIP names to their assigned Qkeys and the function **VStoname** converts Qkeys to DIP names, as shown in Figure 15 and Figure 16.

Figure 15. VStoqkey Input and Output

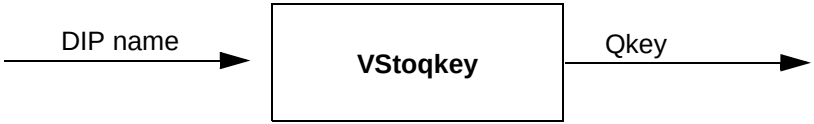


Figure 16. VStoname Input and Output

Figure 17 and Figure 18 on page 156 display the **VStoqkey** and **VStoname** synopsis in C-code for the dynamic DIP.

Figure 17. VStoqkey Synopsis

```
#include <sys/types.h>
#include "VS.h"


key_t VStoqkey(dipName)
char*dipName;/* unique name associated with process */
```

Figure 18. VStoname Synopsis

```
#include <sys/types.h>
#include "VS.h"


char *VStoname(Qkey)
key_tQkey;/* message queue key */
```

For additional information, see VStoqkey on page 619 and VStoname on page 618 in Appendix C, C-Library Functions.

VSerror

VSstartup and **VStoqkey** may return zero or a negative value when an error occurs. At this point, **VSerror** can be called to retrieve a text description of the error. **VSerror** is passed the error value and returns a character string describing the error so that a DIP can log or display the error. Figure 19 on page 157 displays the **VSerror** synopsis written in C-code for the DIP.

Figure 19. VSerror Synopsis

```
#include <sys/types.h>
#include "VS.h"


char *VSerror (errid)
interrid;/* negative error value */
```

For additional information, see VSerror on page 613 in Appendix C, C-Library Functions.

Hardcoded DIPs

startup

The **startup** function is called once to post a hardcoded DIP to the BB. As shown in Figure 20 on page 157, startup takes the Qkey and slot_offset as arguments.

Figure 20. startup Synopsis

```
#include "spp.h"

int startup (qkey,slot_offset)
intqkey; /* Message qkey of calling process */
intslot_offset;  /* used to get slot for posting */
```

In **startup**, the DIP tells the voice system what Qkey the DIP will be using. Hardcoded DIP Qkeys range from DIP0 to DIP34 and, using one of these Qkeys, makes the DIP a message-sending DIP to TSM.

The *slot_offset* argument is used by **startup** to post the DIP in a specific slot in the BB. The *slot_offset* argument is the responsibility of the DIP writer, as it must be known what slots are available for posting hardcoded DIPs. With the increased use of hardcoded DIPs by the voice system, the chance of clashing with other DIPs is possible.

Note: Be aware that other applications may utilize the same hardcoded DIPs causing a clash of resources.

A list of current hardcoded DIPs that the voice system uses is included at the end of this chapter.

Startup computes the slot to post the DIP from the *slot_offset* argument in the following manner:

slot = slot_offset + DIPSTART

DIPSTART is defined in the file **shmemtab.h** as 32. Slots reserved for hardcoded DIPs in the range 32–66, so that the *slot_offset* given should range from 0–34 (the range of DIP numbers). The voice system DIPs use the following convention to compute their corresponding *slot_offset*:

slot_offset = Qkey - DIP0 (where DIP0 is defined as 20 in **mesg.h**)

For additional information, see startup on page 610 in Appendix C, C-Library Functions.

By this point, your DIP should be posted in the BB, and, in the case of a DynaDIP, you should also know the assigned message queue key.

Step 3: Send and Receive Messages

The following sections describe how to send and receive data between DIPs and other processes.

mesgsnd

Once the data is packaged as a message, it can be sent using the C-library function **mesgsnd**. As shown in Figure 21, **mesgsnd** takes a pointer to the message *msgp* of size *msgsz* bytes and sends it to the message queue identified by the Qkey *mdest*.

Figure 21. mesgsnd Synopsis

```
#include <sys/types.h>
#include <sys/ipc.h> |
#include <sys/msg.h>
#include "mesg.h"
#include "spp.h"

int    mesgsnd(mdest,msgp,msgsz,msgflag)
int    mdest;    /* Message Qkey to send to */
union msgunion *msgp;/* message to send */
int    msgsz;    /* size of message */
int    msgflag;  /* flag for controlling send */
```

The *mdest* argument is set to TSM (defined in **mesg.h**) when the DIP wants to send a message to a script running on a channel. The *msgflag* argument is passed directly to the UnixWare system call **msgsnd**, (see the *UnixWare*

Operating System API Reference: System Calls) and is used to determine the actions to take in case of an error. This flag is usually set to zero.

The **mesgsnd** function returns a zero upon successful completion. Otherwise, a negative value is returned. For additional information, see mesgsnd on page 607, in Appendix C, C-Library Functions.

The **mesgsnd** function creates the message queue if necessary.

mesgrcv

The **mesgrcv** function reads the message from the message queue specified by Qkey (*morig*) into a buffer pointed to by *msgp* of size *msgsz* bytes as shown in Figure 22. The **mesgrcv** function creates the message queue if necessary.

Figure 22. mesgrcv Synopsis

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "mesg.h"
#include "spp.h"

int    mesgrcv (morig,msgp,msgsz,msgtyp,msgflg,msgrtime)
int    morig;
char   *msgp;    /* message buffer */
int    msgtyp;   /* type of message to read */
int    msgsz;    /* size of message buffer */
int    msgflag;  /* control flag */
long   *msgrtime;/* message receive time */
```

Since the size of the message varies and is unknown before reading, the message must be read into a buffer that is large enough to accommodate the largest message to be received by the DIP. This assures that all known messages are received properly without being truncated or discarded altogether.

The **mesgrcv** function also allows a DIP to read messages of a particular type (*msgtyp*). The type of message is defined in the field *mtype* in the header **mbhdr**. To read the first message on the queue regardless of its *mtype*, invoke **mesgrcv** with the *msgtyp* argument set to 0 (zero).

The **mesgrcv** function can be used in two places in a DIP. At initialization, it can be used to clear the message queue. Later, it can be used to read requests as they are received. See Appendix A, Application Example for an example.

The *msgflag* field represents a set of flags that control how **mesgrcv** reads the messages. By default, **mesgrcv** waits indefinitely for a message of a specified type to arrive if none are presently on the queue. Many DIPs and voice system processes prefer this because they are message-driven. However, the *msgflag* field allows you specify that you do not want **mesgrcv** to wait for a message to arrive. Currently, the flags are:

- IPC_NOWAIT — If on, mesgrcv returns immediately even if no message has arrived. If off (not specified, or 0), **mesgrcv** *sleeps* until a message arrives.

- MSG_NOERROR — If on, mesgrcv truncates the received message to *msgsz* bytes if necessary.
- IPC_GTIME — If on, mesgrcv returns the UnixWare time (in seconds) when the message was read. The *msgrtime* field must point to a long if IPC_GTIME is specified; otherwise, set it to NULL.

The *msgflag* field is formed by bit applying the Boolean “OR” operation (where the operator is |) to turn on all flags. For example, to not wait for a message and obtain the time the message was read (if any are available), pass the following:

IPC_GTIME | IPC_NOWAIT.

IPC_GTIME and MSG_NOERROR is defined in **mesg.h** and IPC_NOWAIT is defined in **ipc.h**. If no flags are to be turned on, set *msgflag* to zero (0).

The **mesgrcv** function returns the number of bytes read upon successful completion. Otherwise, a negative value is returned. For additional information, see mesgrcv on page 600 in Appendix C, C-Library Functions. For additional information on the UnixWare system call **msgrcv**, see the *UnixWare Operating System API Reference: System Calls.*

Talking to TSM Scripts

Usually, a TSM script initiates the interaction by sending a message to the DIP, which then responds with the information requested. Messages sent by TSM scripts have TSM as the sender and the channel number on which the TSM script is running.

Note: The channel number (*mchan*) must be saved by the DIP for responding to the appropriate TSM script later.

A DIP reads the message using mesgrcv and decides what action to take based on the message id (*mcont*). The message id is set by the TSM script through the second argument to the dbase instruction. Typically, the DIP contains a switch statement on the message id with specific cases for all known message ids, as shown in Sample DIP on page 393 in Application Example on page 386.

DIP Interrupt

Sometimes a DIP initiates the interaction between itself and a TSM script. This is done by sending the DIP interrupt message id defined in **tsm_dip.h**, which interrupts the TSM script instruction currently executing. See Flow Control Instructions on page 70 in Chapter 3, TAS Script Instructions for additional information.

TSM Scripts Talking to DIPs

TSM scripts send and receive messages to and from DIPs through TSM. TSM packages and unpackages the message for TSM scripts. That is, TSM scripts only work with the data part of the message while TSM takes care of either adding or removing the header part, depending on whether the message is sent or received by the script. When sending a message, TSM sends the data from the specified script memory area, and when receiving message, TSM places the data received from a DIP into the specified script memory area.

A TSM script is responsible for:

- Allocating script user memory for holding the largest data being sent or received. Typically, two separate buffers are allocated: one for receiving and the other for sending.
- Inserting the appropriate data into the buffer before sending it to a DIP
- Extracting and accessing the data from the buffer after receiving the data from a DIP

The four functions, **dbase**, **dipterm**, **dipname**, and **dipnum**, are involved with interaction with DIPs. For additional information on these instructions, see Appendix B, Summary of TAS Script Instructions.

- **dbase*(type.dip,mcont_field,ctype.dst,mbyte,type.src,nbyte)***

  Both the sending and receiving of data is done through the **dbase** instruction. The **dbase** instruction first sends the data, then waits for a response from the specified DIP. Responses or messages from DIPs other than the specified DIP are thrown away by TSM. Note that the data received from the DIP, *ctype.dst*, is specified before the data sent to the DIP, *type.src.*

  The **dbase** instruction, when used in the case of DynaDIPs, allows the DIP argument to be the DIP name as well as the DIP number (see **dipnum**). The DIP name is specified using the TSM script language syntax for character strings.

- **dipterm**(***type.dip[,flag]***)

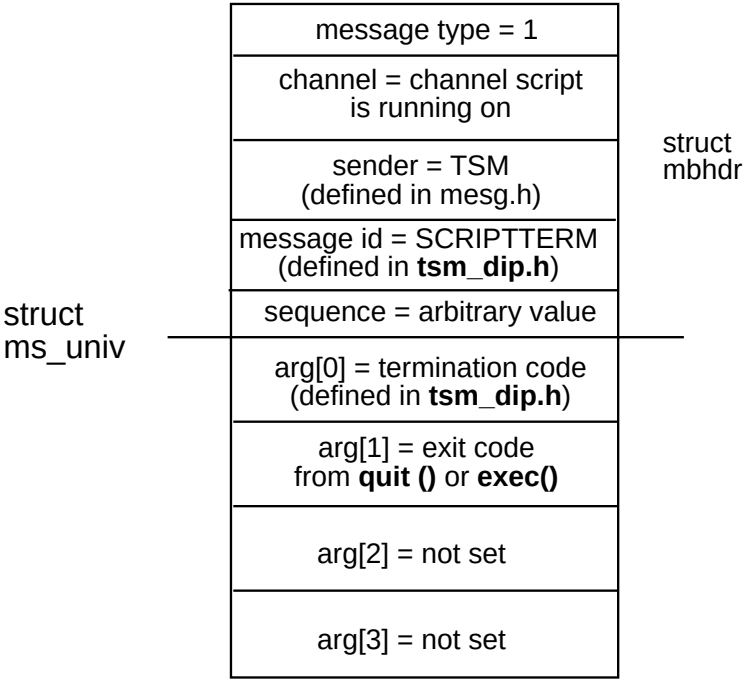
  dipterm instructs TSM to send a message to the specified DIP when the TSM script terminates. As with the **dbase** instruction, **dipterm** allows the DIP argument to be the DIP name as well as the DIP number. The **dipterm** instruction is normally used to perform necessary cleanup after a script terminates.

  The **dipterm** message is defined as the C-structure struct **ms_univ** (see **mesg.h**). Figure 23 and Figure 24 on page 166 show the fields of the message and their values as set by TSM.

Figure 23. dipterm Synopsis

```
/* message structure for dipterm message */
struct ms_univ {
struct mbhdrhd;
longarg[4];
};
```

Figure 24. dipterm Message Structure



The *arg[0]*, as shown in Figure 24 on page 166, displays why the script terminated. As defined in **tsm_dip.h**, there are several causes for a script to terminate.

NORMALTERM	A quit instruction in the script was executed.
DISCONTERM	The call was disconnected.
SCRFAILTERM	An error occurred in the script code.
MTCTERM	The MTC process has seized the channel on which the script is running.
EXECTERM	The script exec'ed another script.

arg[1] is set to the value specified in the quit or exec instructions.

- **dipname**(***ctype.dst,type.src***)

  The **dipname** function takes a DIP number and converts it to the corresponding DIP name.

  The returned DIP name character string is stored in the specified destination address. The destination area should be 16 bytes: 15 characters for the DIP name plus 1 character for the null termination symbol.

dipname is primarily for converting the DIP number returned when a DIP interrupt occurs. This allows scripts working at the DIP-name level to continue by converting the DIP number of the DIP that interrupted them. **dipname** returns a negative value if an error occurs during translation.

- **dipnum**(***type.dst,ctype.src***)

  dipnum converts a DIP name to its corresponding DIP number. **dipnum** returns a negative value if an error occurs during translation.

At this point, you have completed the necessary steps to send and receive messages to the DIP from a TSM script.

Step 4: Implement the Application-Specific Processing

Write the code necessary for the DIP to process the required function and return the desired result. This is the main reason for using a DIP.

Step 5: Define and Add Logger Errors

Define and add logger errors that the DIP will send to the system. This is an optional step, but is recommended. Add as many logger messages as necessary to identify individual problems. For each message, include as much information as possible about variables. These would include return values for important functions and the *errno* value. For more information, see Chapter 6, Message Logger.

Step 6: Add Error Reporting

Add error reporting to notify the logger of errors. This is an optional step, but is recommended. For more information, see Chapter 6, Message Logger.

Step 7: Add Trace Messages

DIPs can be traced by embedding debug information in the DIP and displaying it via the **trace** command and the **db_pr** command. This is an optional step, but is recommended. This feature is enabled by **VSstartup** or **startup**. The purpose of this capability is debugging the DIP during development, rather than providing information about problems in the production application. Include as many **db_pr** statements as are necessary. The debug information should be strategically placed in the DIP code. When the DIP is running, issue the **trace** command from the shell command line to

print debug information on your terminal as it is executed in the DIP code. The **db_pr** statements may be left in while the application is in production. They do not impose a cost or cause any disadvantage.

The trace Command

The trace command allows tracing of specified processes and channels. Trace displays the trace messages on standard out (stdout) that are executed by the specified processes after trace was invoked. For example to trace TSM, channels 0-5, and DIP stock_dip, enter the following at the command line:

trace tsm ch 0-5 stock_dip

Any number of processes and channels can be traced, but only one trace should be running at any one time. Having two trace commands running concurrently causes a sporadic and confusing display of trace messages.

See *UCS 1000 R4.2 Administration,* 585-313-507, for additional information on the **trace** command.

db_pr

The **db_pr** library function applies a variable number of arguments to the format string to form the output trace string as shown in Figure 25.

Figure 25. db_pr Synopsis

```
#include “spp.h”

int db_pr(format, arg ...)
char   *format;  /* printf format string */
```

db_pr trace messages are written to the internal trace buffer and displayed only if tracing is turned on for the corresponding DIP or process. Otherwise, trace messages are ignored while the DIP executes. It is recommended that you use **db_pr** for trace messages because **db_pr** writes to the internal trace buffer only the messages of the processes that currently are being traced. Messages from other processes not being traced are discarded.

Note: Although the **db_pr** structure is identical to the **printf** function, use **db_pr** for DIPs that have user interfaces because it allows a more controlled method of output.

For additional information, see db_pr on page 597 in Appendix C, C-Library Functions.

db_put

The **db_put** library function applies a single string of an argument and displays it as shown in Figure 26.

Figure 26. db_put Synopsis

```
#include "spp.h"

int db_put(string)
char   *string;  /* string to write out */
```

db_put trace messages are displayed when tracing is on regardless of what processes are being traced.

For additional information, see db_put on page 599 in Appendix C, C-Library Functions.

Step 8: Compile and Execute the DIP

The DIP source program is compiled in a standard method using the C-compiler (cc) to include the voice system header files and to link the voice system library libspp.a residing in the directory **/vs/lib**. The voice system header files (**mesg.h**, **VS.h**, **shmemtab.h**) reside under **/att/include**, **/att/msgipc**, and **/usr/spool/log/head**. Whenever a DIP reports errors to the logger/alerter, make sure that _**INSTALLABLE_APPL** is defined (that is, **-D _INSTALLABLE_APPL**).

For example, to create the executable version of DIP stock_dip.c, enter the following:

```
cc -I/att/include -I/att/msgipc -I/usr/spool/log/head \
    -D_INSTALLABLE_APPL -o stock_dip stock_dip.c \
    -L /vs/lib -lspp -llog -lprism
```

Note: This information should appear entirely on one line in the file as indicated by the backslashes (\) at end of the lines.

If your DIP is an IRAPI application, then the compilation line should be changed to the following:

```
cc -I/att/include -I/att/msgipc -I/usr/spool/log/head \
    -D_INSTALLABLE_APPL -o stock_dip stock_dip.c -L /vs/lib \
    -lirEXT -lirAPI -lspp -lTOOLS -llog -lprism
```

For more information about the C-compiler, see the *UnixWare Programming in Standard C and C++.*

Once the executable version is created, you can start it manually from the shell command line or automatically through the inittab file.

Auto Startup Via inittab

A DIP can be started and managed automatically by the UnixWare system process init if it appears in the **/etc/inittab** file. If you display the inittab file, entries for the voice system processes like TSM, logdaemon, and iCk are shown. Typically, one entry is made for each process to run. An entry in the inittab files consists of fields separated by colons (:) that allow you to specify:

1. A unique label to identify the entry (no more than 4 characters)
2. The run-levels to run the program. Voice system processes and most DIPs use run-level 4.
3. Whether the program is to be run once only or re-run if it dies

The **start_vs** command rebuilds the modified inittab file by concatenating all the files in **/etc/conf/init.d**. In order for your entries to be permanent, perform the following procedure:

1. Enter **stop_vs** to stop the voice system.

2. Edit your entry in the **/etc/conf/init.d** directory. The following is an example for a DIP called stock_dip in the **init.d** directory that runs at run-level 4, is re-run if it dies, and is labeled PI1.
   a. Enter **vi stock_dip**
   b. Add the following to the **stock_dip** file.

      PI1:4:respawn:/local/bin/stock_dip > /dev/null 2>&1

   c. Enter **:wq**
3. Enter **/vs/bin/util/mkitab**
4. Enter **start_vs** to start the voice system.

Now your DIP will start up automatically each time the voice system is started. If you experience problems with this procedure, enter **touch /etc/conf/init.d/<*name of the system*>** before restarting the voice system.

Troubleshooting

This section provides some guidelines for detecting problems with your newly created DIP. Although not a thorough treatment, this information gets you started when processes within the voice system appear to be stuck. Debugging a DIP is similar to debugging other C language programs.

- Message queues

  Look for message queues that have unread messages. This may indicate a stuck process or DIP. Use the **ipcs** command to display the current status of the message queues. See the *UnixWare Command Reference* for more information about the **ipcs** command.

- Semaphores

  A semaphore can lock up the system if processes are waiting for its release that never happens. One semaphore is used for each posted process in the BB. Use the **ipcs** command to display the current status of the semaphores. See the *UnixWare Command Reference* for more information about the **ipcs** command.

- Bulletin board

  Use the **bbs** command to display the posted processes in the BB. See *UCS 1000 R4.2 Administration,* 585-313-507, for additional information.

- DIP and TSM scripts

  In **mesgsnd**, **mesgrcv**, or **dbase**, the sender and receiver may be reversed or the DIP may set the mchan value improperly in the return message to TSM. Use the **ipcs** command to display the current status of the DIP. See the *UnixWare Command Reference f*or more information about the **ipcs** command.

- Number of processes

  If you are trying to run a DIP and the system displays a message similar to:

  ```
  cannot fork: too many processes
  ```

  you may need to increase the maximum number of processes that are allowed. See Types of DIPs on page 141 for details on how to tune your system to handle more processes.

- Timeouts

  If there is a long pause in a script near a DIP, it usually means that the DIP did not receive a message, it did not return the message, or it did not return the message properly.

Hardcoded DIPs

Table 16 on page 177 shows the current hardcoded DIPs used in the system. The DIP name and number are listed, as are the software package that interfaces with each DIP. The DynaDIPs are listed as “available,” meaning they are not used to interface solely with one software package. Also, be aware that some of those marked available are actually called by other packages. If you do not have a certain package installed on your system, that hardcoded DIP will not be occupied.

Table 16. Hardcoded DIPs

DIP Name	DIP #	Package
agdip3270	DIP0	Host
oraldb	DIP1	Local Database
reserved	DIP2	
RPTDIP	DIP3	FAX Actions
Spadm DIP	DIP4	Speech Admin
reserved	DIP5	
reserved	DIP6	
asaihp	DIP7	ASAI
reserved	DIP8	
ADMDIP	DIP9	FAX Actions
xferdip	DIP10	Call Bridge
agdiphelper	DIP11	Host
	DIP12	available
	DIP13	available

1 of 3

Table 16. Hardcoded DIPs

DIP Name	DIP #	Package
	DIP14	available
FAXDIP	DIP15	FAX Actions
FAXMGR	DIP16	FAX Actions
	DIP17	available
	DIP18	MTC
	DIP19	VROP
	DIP20	available
reserved	DIP21	
	DIP22	available
	DIP23	available
reserved	DIP24	
FAXCNG	DIP25	FAX Actions
dc.sh	DIP26	Chan 0
		Data Collection

2 of 3

Table 16. Hardcoded DIPs

DIP Name	DIP #	Package
dc.sh	DIP27	Chan 1
		Data Collection
dc.sh	DIP28	Chan 2
		Data Collection
dc.sh	DIP29	Chan 3
		Data Collection
dc.sh	DIP30	Chan 4
		Data Collection
reserved	DIP31	
	DIP32	Voice
		Workstation
	DIP33	available
	DIP34	available

3 of 3

TTS_DIP

Note: For the UCS 1000 R4.2, the TSM fsay instruction provides the same functionality as what is described for tts_file.

There are many reasons a DIP may be needed in an application using Text-to-Speech (TTS). A simple, general DIP is provided with the TTS software to give an idea of what should be included in a custom DIP. This generic DIP is called **tts_dip**.

The **tts_dip** supports the following functions:

- Read file — Reads a given number of bytes of ASCII text sent via IPC messages and returns to the calling process (TSM or any other process). The maximum read is 512 bytes. The default directory for this ASCII file is **/vs/data/tts_files**. By providing an absolute path, a script can override this default directory.
- Reset file — Frees all per-channel space for this file and channel.
- Script termination — If **tts_dip** receives this message from a valid channel, it stops all activity for that channel and cleans up the per-channel space.

Message Interfaces with tts_dip

The message interfaces with **tts_dip** are defined in **tts_iface.h**. The following messages are accepted by **tts_dip**:

- READ_FILE — Read from an ASCII file
- RESET_FILE — Reset the file
- SCRIPTTERM — Script termination message (defined in **/att/msgipc/tsm_dip.h**)

For READ_FILE and RESET_FILE messages, the following message structure is used:

```
struct msg1 {
  struct mbhdr hd;
  int  start;
  int  to_read: /* Number of bytes to read. Max. 512 bytes. */
  char    file{F_NAME_LEN}; /* Max. name length if 128 bytes to
*/
                /* allow for absolute path for a file */
                /* name. */
};
```

Fields *to_read* and *start* are not used for the RESET_FILE message. The to_read field should be set to the number of bytes to be read, up to a maximum of 512 bytes. For READ_FILE, the start field should be set to 0 to read from the beginning of the file or to 1 to read from the current position in the file. The **tts_dip** does not modify the contents of the *mtype* and *mseqno* fields.

For successful READ_FILE reply messages, the following message structure is used:

```
struct msg2 {
  struct mbhdr hd;
  inst Ret_len; /* Number of bytes read */
  char bytes[MAX_READ+1]; /* Actual ASCII text bytes. */
};
```

The field *mcont* in the header is set to R_PASS. The *Ret_len* field indicates how many bytes were read (this may be less than the requested number of bytes). If the *Ret_len* value is less than the *to_read* value, the end of the file was reached during the current READ_FILE request.

The following message structure is used for all other message to and from the **tts_dip**:

```
struct ms_univ { /* struct defined in mesg.h */
  struct mbhdr hd;
  long    arg[4];
};
```

No acknowledgment is sent for SCRIPTTERM and RESET_FILE messages. When one of these messages is received, the **tts_dip** cleans up the per-channel space for the channel and closes the file used for that channel.

If a READ_FILE message fails, the return message *mcont* is set to R_FAIL and *arg[0]* is set to:

- -1 – ASCII text file not found
- -2 – Invalid argument

Message structures and mconts are defined in **/att/include/tts_iface.h**.

5 IRAPI

Overview

This chapter provides information about the Intuity™ Response Application Programming Interface (IRAPI). This information includes:

- How the IRAPI is organized in relation to the rest of the platform, terminology associated with the IRAPI, and a system-level architectural description.
- 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).
- The run-time services available through the IRAPI. An example IRAPI application (chantest) is presented, which 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 and provides a high level description for each function used. The entire chantest application is available on the system in the file **/vs/examples/IRAPI/chantest.c**.

- Details of the IRAPI functions and data structures and how applications can be built using them. Functional areas are grouped 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.
- The steps necessary to compile and install an IRAPI application on the UCS 1000 R4.2. The compile and install procedure uses the chantest.c application, available on the system as an example.
- The various tools that are available to an application developer when trying to debug an IRAPI application.
- The resource management performance issues for IRAPI applications, including a list of the Resource Manager (RM) driver tunable parameters for the system. These parameters control the RM driver's capacity and behavior.

Introduction to the IRAPI

This section 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
- Dial pulse and speech recognition
- Text-to-Speech (TTS) processing
- Resource management
- H.110 bus management (bridging and monitoring channels)

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

Manual Pages for Commands and Parameters

IRAPI manual pages (commonly called man pages) are available on the system. The section one (1IRAPI) manual pages, describing IRAPI related command line utilities are stored in **/vs/man/cat1**. The section three

(3IRAPI) manual pages, describing IRAPI library subroutines are stored in **/vs/man/cat3**. The section four (4IRAPI) manual pages, describing IRAPI file formats, define symbols, events, parameters, resources, and states are stored in **/vs/man/cat4**.

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 UnixWare 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 isolated 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, voice file descriptors (which are very similar to UnixWare file descriptors), or UnixWare files. Speech is stored in standard UnixWare filesystems. In order to support applications and packages that rely on storing phrases in the filesystem, the IRAPI supports mapping talkfile and phrase numbers to UnixWare files and vice versa.

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 UnixWare 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 speech and signal processor (SSP) 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 for custom speech provided by the application developer. Applications have a similar interface and level of control over TTS activities.

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, Dial Pulse Recognition, and Speech Recognition

Touch tones are collected in a unified input queue that can be manipulated in a variety of ways. The same input queue is used for touch tone, dial pulse recognition (DPR), and speech recognition input. 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.

Timeslot Management

The IRAPI provides functions for managing the H.110 bus and network interface connections to the bus. Applications running on several channels can bridge their H.110 bus 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.

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 default

owner listening for new calls and dispatching applications in response to them (see Application Control on page 209). 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 UnixWare **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 SSP circuit cards. The IRAPI library is linked to processes that use the IRAPI. The library uses the UnixWare kernel, device drivers for SSP 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. Scripts use the IRAPI to manage speech pools. Other applications that use the IRAPI might be user processes, custom applications or conventional UCS 1000 R4.2 data interface processes (DIPs).

Figure 27 on page 196 shows how the IRAPI is organized in relation to the system software processes. Figure 28 on page 197 shows how the IRAPI fits into the voice system architecture. 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.

Figure 28 on page 197 shows several processes that use the IRAPI facility. Only the relationships between processes that are relevant to the IRAPI are shown in Figure 28 on page 197. For example, all of the voice system processes invoke the logger.

- 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.

- TSM

  TSM is a multi-channel IRAPI application that runs TAS scripts.

- Application Dispatch (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 27. IRAPI Organization

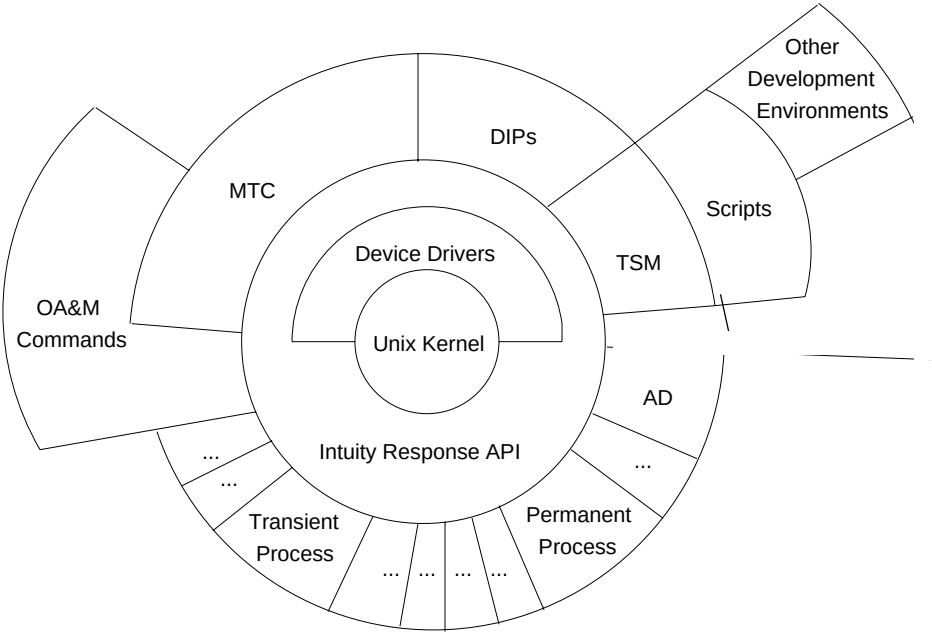
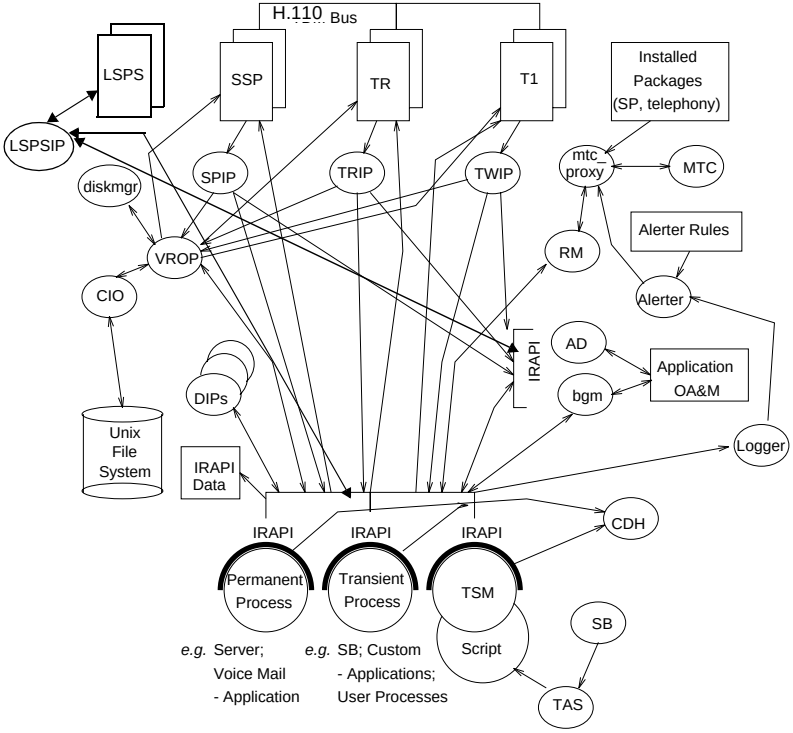


Figure 28. IRAPI Architecture Overview



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 work together 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 UnixWare 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 always forcing the applications to re-compile. If applications are required to be re-compiled, then application developers will be notified in release notes.
- In the IRAPI, application control and resource management is distributed over many processes in the system; the IRAPI coordinates all the applications.

  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 UCS 1000 R4.2 system logger. The logger accepts the error message and, among other things, forwards it to the UCS 1000 R4.2 system 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. Dynamic resources are resources that are not associated with a channel (for example, SSP 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 (**rmdb**) can be used to query the state of the managed resources.

RM also manages the messages sent to and received from all IRAPI processes. Mesgrev and mesgsnd calls go through the RM process. Use the **/vs/data/unix_ipc_qkeys** file to specify which qkeys should use the UNIX IPC messages.

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 SSP circuit card) are maintained by the MTC platform.

Application Dispatch (AD)

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. See Application Dispatch API on page 210. 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.

Note: You can administer the DNIS and ANI tables using the **assign, delete,** and **display** commands. See Appendix A, "Summary of Commands" in *UCS 1000 R4.2 Administration*, 585-313-507, for additional information about these commands.

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 (for SSP or Tip/Ring hardware) or LSPSIP (for LSPS II hardware) process. Plays or codes for SSP or Tip/Ring hardware can use memory buffers, entire UnixWare files or UnixWare file descriptors as sources and destinations. Plays or codes for LSPS II hardware use entire UnixWare files. The VROP process uses the customer input/output (CIO) processes to read and write information in and out of the UnixWare filesystem. In order to actually play or code speech, the VROP process interacts with the SSP or Tip/Ring. Similarly, the LSPSIP process interacts with the LSPS II card to actually play or code speech. For language playback (LP), the IRAPI uses the tables to associate recorded phrases with vocabulary items to support the process of speaking language-independent alphanumeric strings, numbers, dates, times, and currencies.

IRAPI with UCS 1000 R4.2 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, Line-Side T1 (LST1), and PRI. 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 can be irExec'ed or irSubProg'ed from IRAPI applications much like the same scripts could be exec'ed from other TAS scripts.

Note: The LSPS II circuit card supports no more than 64 calls to irSay(3IRAPI) or irPlay(3IRAPI) functions. The application must call the irEnd(3IRAPI) after 64 irSay(3IRAPI) or irPlay(3IRAPI) calls, or unexpected results can occur.

Application Organization

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

- Use 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

TAS makes 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 UnixWare 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 irExec or irSubProg 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 started. 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 the voice system bulletin board 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 UnixWare executable programs 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 data not initialized data in the program.

Note: Since data and BSS areas can be changed at run time, 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 UnixWare 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 approximate space used by the IRAPI library is shown in Table 17.

Table 17. Memory Usage

Library	Text	Data	BSS
/usr/lib/libirAPI.so	1099252	133872	14140
/usr/lib/libirEXT.so	1 000	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-channel or single-channel processes.

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

Single-channel vs. Multi-channel Permanent vs. Transient

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

- Permanent processes are loaded and initialized at system startup time; transient processes are loaded and initialized on **irExec** or **irSubProg**.
- Permanent processes are quicker to load and start processing user inputs Permanent process also use CPU resources more efficiently.
- Single-channel processes can mix **irWait()**'s throughout their program. This is impossible for multi-channel applications.

- Transient processes must deal with creating unique names.
- Permanent processes may want to clean out their message queue as they start (as shown below). Transient processes do not want to miss an IRE_EXEC message.

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

the application characteristics between single-channel and multi-channel applications and between permanent and transient processes are compared in Table 18.

Table 18. Application Characteristics

	Single Channel	Multi-Channel
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

This section 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-channel 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 initializes 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 an 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 irExecs the channel to the startup application. The startup application performs the special processing and then either irExecs 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 ECS or compatible switch. 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 or irSubProg functions 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** command (see below).

defService command

The **defService(1IRAPI)** 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 Appendix A, "Summary of Commands," in *UCS 1000 R4.2 Administration*, 585-313-507. For TSM scripts, the output of the **tas** command serves as the service registration file for the script.

If it is 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

This section discusses the run-time services available through the Intuity Response Application Programming Interface (IRAPI) for the UCS 1000 R4.2. 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 is available on the system in the file **/vs/examples/IRAPI/chantest.c**.

This section 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.

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 UnixWare 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 multi-channel 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\\\n");
    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) or irSubProg(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 the online instructions on the system about IrEVENTS(4IRAPI)].

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. See Channel Management on page 240.

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 the online instructions on the system about irEvent(3IRAPI) and IrEVENTS(4IRAPI). 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_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);
}
```

4 Set any IRAPI parameters to be used by the application with irSetParam or irSetParamStr [see the online instructions on the system about irParam(3IRAPI)].

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 delineator for input shorter than IRP_INPUT_LEN.

Application Execution

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 onirAnswer",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);
    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);
}
```

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 UnixWare 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);
    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);
    }
}
```

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);
    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 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 “good-bye" 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.

If an irSubProg(3IRAPI)’ed process wants to return channel ownership back to a parent process, it should call irReturn on the channel_id. Once irReturn has been called, the cid value originally obtained for the channel through irInit or irInitGroup 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 “good-bye" 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) or irReturn(3IRAPI).

Process Termination

IRAPI processes should irDisconnect(3IRAPI) and irDeinit(3IRAPI) or irReturn(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)** UnixWare 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

- Dial pulse and speech recognition
- Resource management
- Text-to-Speech (TTS)
- Fax
- 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 (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 the online instructions on the system about irChan2Cid(3IRAPI) and irCid2Chan(3IRAPI)].

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 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)'ed to that process. After being irExec(3IRAPI)'ed, the IRP_SERVICE_NAME parameter is set to the value indicated during service definition [see **defService** in Appendix A, "Summary of Commands," in *UCS 1000 R4.2 Administration*, 585-313-507].

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 executes or “execs” 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 to 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.

SubProging Applications on Channels

A process subprog's another process on a channel via an irSubProg(3IRAPI) function. Channels may be irSubProg'ed to permanent or transient processes. Permanent processes are typically multi-channel applications run from inittab. Transient processes are **exec(2)**'ed by the irSubProg 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 irSubProg(3IRAPI). Once an irSubProg function is called, the calling process must stop using the channel_id. Channel ownership is made available immediately to the irSubProg'ed application. After receiving the IRE_EXEC event, the channel should be initialized immediately via irInit(3IRAPI). Once the called process gives up control of the channel with irReturn(3IRAPI), then channel ownership reverts back to the parent process. The parent process may use the channel_id again.

As with irExec, parameters may be used to allow the child process to run in a context similar to the parent process. The discussion in the previous section on the IRP_EXEC_BUF and IRP_REGISTER parameters applies to irSubProg and irExec.

Execing TAS Script Applications

The following parallels exist between the IRAPI and TAS scripts:

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

  Any TAS script irExec'ed or irSubProg'ed 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 TAS script registers r.0 through r.15

  The TAS 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.

See Chapter 3, TAS Script Instructions 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 (or irSubProgp) 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 (see Chapter 3, “Voice System Administration,” in *UCS 1000 R4.2 Administration*, 585-313-507). 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'ed or irSubProg'ed to an application, the channel is, in effect, unowned for a short period of time. The processes receiving the IRE_EXEC event (the irExec'ed or irSubProg'ed 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) or irReturn(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.c) uses irInit(3IRAPI) to seize an idle channel. See the sample application on the system.

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 of an irExec'ed process. 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) or irReturn(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). See Execing Applications on Channels on page 242 for additional information.
- Calling irSubProg(3IRAPI) to pass ownership to another process — Renders the channel_id temporarily invalid (for the duration of the process being irSubProg'ed) and removes channel ownership from the calling process after a successful return from irSubProg(3IRAPI). See SubProging Applications on Channels on page 244 for additional information.
- Calling irReturn(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 irReturn(3IRAPI). See Execing Applications on Channels on page 242 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.

Exec vs. Subprog

In summary, for a transient child process both irExec and irSubProg perform a **fork(2)** and **exec(2)**.

In the case of irExec, channel ownership goes from the parent process to the child process. When the child process relinquishes channel ownership the default owner then owns the channel.

In the case of irSubProg, channel ownership goes from the parent process to the child process. When the child process relinquishes channel ownership the parent process resumes ownership of the channel and may continue to use the channel_id associated with the channel. The child process returns ownership to the parent via irReturn.

For a permanent child process neither irExec nor irSubProg perform a **fork(2)** and **exec(2)** as the permanent child process is already running. The permanent child process was started via **/etc/inittab** or similar mechanism.

Channel ownership behavior as described above with transient child processes is the same with permanent child processes. TAS scripts may irExec or irSubProg IRAPI applications via the **exec** or **subprog** script instructions. IRAPI applications may irExec or irSubProg TAS scripts via the irExec or irSubProg functions. The irExecService or irSubProgService functions are especially useful here.

The UnixWare tunable parameter called NCONTEXTSTACK defines the maximum number of irSubProg'ed processes available to a single IRAPI application. The default value of this parameter is 3; the minimum value is 1 and the maximum value is 10. If more than NCONTEXTSTACK processes are irSubProg'ed, then the return from irSubProg is -1 and the value of irError is set to IRER_SYSERROR and irSysError is set to ENOMEM.

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 see 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_CALLINTR — Interrupt calling
  - IRF_DEINIT — Interrupt all activity and deinit
  - 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_SUBPROG_NOTIFY — Pass this event back up to the parent

- 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.

Polling with Stream-Oriented Devices

irWait is compatible with UnixWare signals in that irPoll(3IRAPI) functions can be used on stream-oriented devices. These functions may be used to multiplex input from irWait(3IRAPI) and other file descriptors.

```
int handle_poll(int fd, short revents)
   {
      /* handle your input here */
   }
   main()
   {
      ...
      irAddPoll(fd, POLLRDNORM|POLLHUP, handle_poll);
      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. See the following code fragment for an implementation example

```
main ()
{
      inttag=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 UnixWare 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 in Table 19 on page 265.

Table 19. Channel-Specific Parameters

Parameter Attribute Name	Description
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 that are saved across a call to irExec(3IRAPI) or irSubProg(3IRAPI). These parameters 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.
read-only	Indicates whether the paramater can or cannot be changed by the application.
auto-init	Indicates whether IRAPI automatically initiates the parameter.

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

When a channel is released via irDeinit(3IRAPI) or irReturn(3IRAPI), most 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 5, “Switch Interfaces,” in *UCS 1000 R4.2 Administration,* 585-313-). 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, available on the system in the file **/vs/examples/IRAPI/chantest.c**, 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 the online instructions on the system about irIE(3IRAPI) 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 UnixWare 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 on the SSP circuit card only). 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.

The irSpeakChar() function may be used to queue a character string for play. 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 and is only supported for speech on the SSP circuit card.) 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 [seeirPARAMETERS(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). This function is only supported by speech to be played on the SSP circuit card.
- 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. This function is only supported by speech to be played on the SSP circuit card.
- 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). This function is only supported by speech to be played on the SSP circuit card.
- 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 on the system 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 on the system 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).

Note: CCA is only supported onthe SSP circuit card.

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 the online instructions on the system about irServiceState(3IRAPI). 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)'ed or irSubProg(3IRAPI)'ed 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, intelligent, Full CCA)
- 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), irDeinit(3IRAPI), or irReturn(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 available on the system in the file **/vs/examples/IRAPI/chantest_oc.c**. Also, included on the system is the mkcall sample application (in the file **/vs/examples/IRAPI/mkcall.c**) 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 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_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_TIMEOUT:
    case IREM_FAST_BUSY:
    case IREM_ERROR:
    default:
     if (irDisconnect(cid, 0) < 0) {
         cleanup ("Error on irDisconnect", cid);
     }
     break;
  }
  break;
```

Using Call Classification Analysis

Note: CCA is only supported onthe SSP circuit card.

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. For accurate transfer results, assign Full CCA only to an SSP circuit card (see Chapter 3, “Voice System Administration,” in *UCS 1000 R4.2 Administration,* 585-313-507).

The following parameters affect CCA:

- IRP_OUTCALL_CCALEVEL — Determines the CCA level (blind, intelligent, Full CCA); 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 IRAPI functions that control the timeslots on the H.110 bus on the system.

The H.110 bus is used in the system to transfer speech data between E1, T1, SSP, and LSPS II circuit cards. There are 4096 communication paths, called timeslots, on the H.110 bus. To transfer speech data between cards, an IRAPI application must reserve a H.110 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 inputting 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) or irReturn(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. Speech to be played on the LSPS II circuit card must be recorded at lower levels as the LSPS II circuit card does not support gain control.

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 H.110 timeslots can be enabled or disabled by setting the IRP_DTMF_MUTING parameter. See the online instructions on the system about irPARAMETERS(4IRAPI) 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 irStop(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 — (Only supported for speech played and recorded on the SSP circuit card.) Determine the coding algorithm type of a voice object. A voice object is a UnixWare file or a program memory buffer containing speech data.
- Algorithm conversion — (Only supported for speech played and recorded on the SSP circuit card.) 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 UnixWare files — Convert talkfile phrase id pairs to UnixWare 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 see the algorithm type.

Voice File Descriptors

Voice file descriptors are similar to UnixWare file descriptors. They are used by IRAPI applications to position the voice file pointer to arbitrary points, measured in milliseconds (for the SSP circuit card), 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 on the SSP circuit card, 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_PLAY_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

Note: The information in this section only applies to speech played and recorded on the SSP circuit card.

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);
}
```

Note: A voice file coded with the IRA_CELP16 code style requires an SSP circuit card.

Algorithm Detection

Note: The information in this section only applies to speech played and recorded on the SSP circuit card.

The irGetAlgorithm(3IRAPI) function returns the algorithm type of the voice file associated with a voice file descriptor (see the example undert Voice File Positioning and Speech Headers on page 291). Algorithm detection functions also exist to determine the encoding type of a voice file given a UnixWare path name [irFGetAlgorithm(3IRAPI)], or a buffer [irBGetAlgorithm(3IRAPI)]. The command line utility codetype(1) may be used to determine the coding type for a voice file.

Algorithm Conversion

Note: The information in this section only applies to speech played and recorded on the SSP circuit card.

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 expensive to compute. 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 UnixWare 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 UnixWare 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 system stores speech in standard UnixWare files. This allows the use of standard UnixWare commands to manage speech data.

The system voice file system commands, the UnixWare equivalents, their status, and a brief description of the command's function are shown in Table 20. For additional information on the voice commands, see Appendix A, "Summary of Commands," in *UCS 1000 R4.2 Administration,* 585-313-507. For additional information on the UnixWare commands, see the *UnixWare Command Reference*.

Table 20. Voice File System Commands

Voice Command	Description	UnixWare Analogue
vdf	Find used/usable space in filesystem	df
copy	Copy speech phrases	cp, cpio
audit	Audit a filesystem	fsck
buildfs	Build a raw slice speech filesystem	mkfs
addhdr	Add voice or code header to a speech file	-

1 of 2

Table 20. Voice File System Commands

Voice Command	Description	UnixWare Analogue
list	List phrases in a speech slice	ls
erase	Remove (erase) speech phrases	rm
add	Add speech to a filesystem	cp, cpio
spsav	Save speech to a tape	-
spres	Restore speech from a tape	-
striphdr	Removes speech coding headers	-

2 of 2

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

Dial Pulse and Speech Recognition

Dial pulse and speech recognition provide caller input to an application. Like touch-tone input, 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.

Dial pulse and speech recognition differ 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.

WholeWord speech recognition works closely with echo cancellation to support barge-in (recognition during prompting) and talkoff (prompt stops playing when speech is first heard). 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.

See *UCS 1000 R4.2 Speech Development, Processing, and Recognition,* 585-313-212, for additional information about speech recognition. See the maintenance book for your platform for information on the software packages that must be installed to support dial pulse and speech recognition (Dial Pulse Recognition (DPR), WholeWord speech recognition, and FlexWord speech recognition).

Recognition Functions

The following functions are provided to support dial pulse and speech recognition:

- irInitRecog(3IRAPI)
  - ~ Initialize recognition parameters per IRP_RECOG_TYPE
- irInitRecog2(3IRAPI)
  - ~ Initialize recognition parameters per *recog_type*
- irInitAllRecog(3IRAPI)
  - ~ Initialize parameters for all recognizers available
- irStartRecog(3IRAPI)
  - ~ Start recognition per IRP_RECOG_TYPE, IRP_RECOG_GRAMMAR, and IRP_RECOG_GRAMNAME
- irStartRecog2(3IRAPI)
  - ~ Start recognition per *recog_type* and *recog_grammar* (on SSP circuit card only)
- irStopRecog(3IRAPI)
  - ~ Stop recognition per IRP_RECOG_TYPE
- irStopRecog2(3IRAPI)
  - ~ Stop recognition per *recog_type*

- irStopAllRecog(3IRAPI)
  - ~ Stop all started recognizers
- irCheckRecog(3IRAPI)
  - ~ Check the on/off status of recognition per IRP_RECOG_TYPE
- irCheckRecog2(3IRAPI)
  - ~ Check the on/off status of recognition per *recog_type*
- irStartRecogTimer(3IRAPI)
  - ~ Start the recognition timer per IRP_RECOG_TYPE
- irStartRecogTimer2(3IRAPI)
  - ~ Start the recognition timer per *recog_type*
- 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. See the online instructions on the system for additional information about 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
  - ~ SSP circuit cards recognition types: IRD_DIALPULSE, IRD_WHOLE_WORD, or IRD_FLEX_WORD
  - ~ LSPS II circuit card recognition types: IRD_LSPS_WHOLE_WORD, LRD_LSPS_FLEX_WORD
- IRP_RECOG_GRAMMAR — Depends on IRP_RECOG_TYPE, valid values are found with the grammar header files associated with the recognition type (supported using the SSP circuit card only)
- IRP_RECOG_GRAMNAME — Specific the recognition grammar name for WholeWord and FlexWord speech recognition (supported using the LSPS II circuit card)
- IRP_RECOG_PRETIME — Specifies a timeout to wait for caller input, measured in milliseconds

- IRP_RECOG_INTERTIME — Specifies a timeout to wait for caller input between digits, measured in milliseconds [applies only to touch tone (DTMF) and dial pulse recognition]. See IrPARAMETERS(4IRAPI) and the information below for the new timer parameters for LSPS II speech recognition.
- IRP_ECHOCAN_TYPE — Currently only two supported types: IRD_SP_ECHO and IRD_LSPS_ECHO (for the SSP and LSPS II circuit cards, respectively)

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

The following parameters are available for LSPS II speech recognition only.

- IRP_RECORD_INIT_EFLOOR — Specifies the initial speech energy floor in decibels indicating silence for speech recording
- IRP_RECORD_END_EFLOOR — Specifies the ending speech energy floor in decibels indicating silence for speech recording
- IRP_RECOG_EFLOOR — Specifies the speech energy floor in decibels indicating silence for speech recognition
- IRP_RECOG_GAPTIME — Specifies the number of milliseconds to assume between speech utterances
- IRP_RECOG_MIN_UTIME — Specifies the minimum acceptable length in milliseconds of a speech utterance to be considered valid speech

- IRP_RECOG_MAX_UTIME — Specifies the maximum acceptable length in milliseconds of a speech utterance to be considered valid speech
- IRP_RECOG_MAX_LTIME — Specifies the maximum length in milliseconds the speech recognizer will listen

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

Dial Pulse Recognition (DPR), WholeWord speech recognition, and FlexWord speech recognition work similarly 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 types of recognition is that barge-in is not supported with DPR or FlexWord recognition. However, the IRAPI does not prevent you from attempting to use the echo canceler or recognize during prompting with DPR or FlexWord recognition; 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.

Note: Although IRAPI handles recognition in similar ways, the SSP and LSPS II circuit cards handle speech recognition differently.

Grammars

- Creating a grammar for the LSPS II circuit card

  Use the following procedure to create a grammar for recognition on the LSPS II circuit card:

  [1] Create a ***<name>***.g file in the directory **/vs/lsps/asr/grammar/*<ASRNAME>*/gramfile**. See Chapter 13 LSPS Software Development Kit resident on the system at **/usr/lsps/doc/pdf/sdk.pdf**

Note: You must access the **sdk.pdf** on a system that contains Adobe Acrobat Reader.

[2] Enter **gc_make *<name>*** where *<name>* is the name of the grammar.

[3] Enter **mkboardtype -f -t *<name>*** where name is one of the following:

CS_BASIC_ASR_FLEX-6UB5 (Whole Word and FlexWord)
CS_BASIC_ASR_FLEX_TTS-6UB5 (Whole Word, FlexWord, and TTS)
CS_BASIC_FLEX-6UB5 (FlexWord only)
CS_BASIC_FLEX_TTS-6UB5 (FlexWord and TTS only)

[4] Update the USAGE0, MAX_CONCURRENT, and MAX_WITH_TTS parameters within **/vs/cslsps/usage/flexword** usage file.

- Header files for SSP speech recognition
  - ~ Dial Pulse Recognition

    Dial Pulse Recognition grammars are defined in the header file:

    /att/asr/grammar_hs/dpr.gram.h

    There is one Dial Pulse grammar file for all countries. Applications using Dial Pulse Recognition would include the grammar header file into their programs as follows:

    #include "/att/asr/grammar_hs/dpr.gram.h"

  - ~ WholeWord Recognition

    WholeWord recognition grammar 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 speech recognition would include the grammar header file into their programs as follows:

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

~ FlexWord Recognition

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 delineators settings do not effect the generation of the IRE_INPUT_DONE event in this case.

Dial Pulse Recognition (DPR) uses IRE_INPUT_DONE similarly to speech recognition. DPR also uses the IRE_RECOGNIZING event. This event is reported by DPR as soon as it detects dial pulse input. This event can be used to turn off other simultaneously running recognizers. The IRE_RECOGNIZING event is not presently available for speech recognition.

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 SSP or LSPS II circuit card. This is only relevant for Tip/Ring channels set to talk

(that is, voice output is played on the Tip/Ring circuit card rather than an SSP circuit card). If any such channels exist, the IRAPI automatically switches the channels to tdm (meaning voice output is played on an SSP circuit card). Talk and tdm correspond to IRP_VOICE_TYPE settings of IRD_TALK_VOICE, IRD_SP_VOICE, and IRP_LSPS_VOICE respectively. When using echo cancellation, the VOICE service must be assigned to an SSP or LSPS II circuit card in the system. See Chapter 3, “Voice System Administration,” in *UCS 1000 R4.2 Administration,* 585-313-507, to assign service to a circuit card.

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

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

In general, applications that use echo cancellation consume more SSP or LSPS II 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. The entire chantest_arc.c application is available on the system in the file **/vs/examples/IRAPI/chantest_arc.c**.

chantest_asr.c must 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 ( 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;
```

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 increased 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 Tip/Ring 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 SSP and LSPS II cards. Examples of dynamic resources include voice play and record resources on SSP and LSPS II 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, if licensed resources are available. Resources are licensed on a specified number of channels for each feature. The following functions implicitly allocate resources:

- irStartEcho(3IRAPI) — Allocates echo cancellation resources
- irStartRecog(3IRAPI) — Allocates recognition resources
- irEnd(3IRAPI) — Allocates SSP or LSPS II play or TTS resources depending on whether play or say requests have been queued
- irRecord(3IRAPI) — Allocates SSP or LSPS II 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 licensed 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. See the online instructions on the system 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 licensed resources for a feature 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 licensed 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_FAX — Fax send/receive
- IRC_RECOG — Input 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_LSPS_VOICE, an LSPS play resrouce is allocated. If IRP_VOICE_TYPE is set to IRD_SP_VOICE, an SSP 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) and irSubProg(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 SSP cards and 1 VOICE SSP 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 under utilized 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) and irSubProg(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 SSP LSPS II circuit cards are put into service. The usage vector, described in the following section, describes the load the resource usage imposes on the SSP or LSPS II circuit card.

RM uses an algorithm whereby the least functional, lightest loaded SSP or LSPS II circuit card is selected. For example, if a system contains 2 SSPs, 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 SSP 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 35801deferPid -1   pendingForce 0    pendingForceDev 0
    profile ptr 0xd140600 work 0x00000000
    timerID 397267  nTimers 1
    Allocated List:  empty
    Restricted List:  empty
    Pending List:
        SP_PLAY(0) tag 0x1card -1allocNo 0x0     next 0x0
    Change Own Pending List:  empty
    Context stack:  size:  -1   pointer 0x0
    Uninitialized context stack
  51  :ownDev 10  type 1   ownQ 120
    defOwn 35801deferPid -1   pendingForce 0    pendingForceDev 0
    profile ptr 0xd1403000 work 0x00000000
```

```
timerID 397020  nTimers 1
Allocated List:
    SP_PLAY(2) tag 0xfeedcard 4allocNo 0x20000next 0x0
Restricted List:  empty
Pending List:  empty
Change Own Pending List:  empty
Context stack:  size:  -1   pointer 0x0
Uninitialized context stack
```

rmdb also can be used to check the resource utilization of SSP circuit card. The following example uses **rmdb -c14** 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 resources such as SP_WW_RECOG and SP_PLAY.

```
rmdb -c 14
Card table:
14  :  MTC_INSERV     2          0x00000000
     0  "SP_RECORD          " [83      0      0      0    ] [ 0/0 /120]
     1  "SP_ECHOCAN         " [150     0      0      0    ] [ 0/0 /60]
     2  "SP_PLAY            " [83      0      0      0    ] [ 0/0 /120]
     3  "SP_TTS              " [166     0      0      0    ] [ 0/0 /60]
     4  "SP_WW_RECOG        " [600     0      0      0    ] [ 0/0 /15]
     5  "SP_CELPCODE        " [166     0      0      0    ] [ 0/0 /60]
```

```
Saturated     : [10000  10000   10000   10000 ]
High Water    : [0      0       0       0     ]
Current       : [0      0       0       0     ]
Allocation #  : 0x0     0x0     0x0     0x0
```

SSPs and LSPS IIs are complex resources made up of multiple hardware and software components. Utilization of SSP and LSPS II resources by a function cannot be described with a single value. Usage vectors allow for the description of SSP and LSPS II 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 SSP and LSPS II. 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 SSP and LSPS II 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 SSP and LSPS II 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 may run either on the CPU, the SSP, or the LSPS II. CPU supported TTS requires a fast CPU (120 MHz or faster) and installation of the system software Text-to-Speech package. Hardware assist TTS requires one or more SSP or LSPS II circuit cards capable of running TTS and the installation of the Text-to-Speech software package. See *UCS 1000 R4.2 System Description*, 585-313-209, and the maintenance book for your platform for additional information about the hardware and software requirements for the TTS packages.

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 voice file descriptor. This file descriptor is obtained from irOpen(3IRAPI).
- 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 (only supported with the SSP circuit card) 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) function cannot be used for TTS. The irGetVCount(3IRAPI) function can 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.

FAX

The following information applies to IRAPI function calls when implementing fax capabilities. See the fax examples in Appendix A, Application Example.

Note: Only the functions listed in this section are currently supported when implementing fax on the SSP circuit card.

Sending a Fax

- Use irFAXPrint, irFAXPrintText, or irFAXBPrintText to queue one or more files.
- Use irFAXEnd to send the files.
- Ignore the IRE_FAXPRINT_PROG event when logged and wait for IRE_FAXPRINT_DONE.

Note: Note the FAX streaming mode (queueing additional files after calling irFAXEnd) is an obsolete feature and should be avoided.

Receiving a Fax

Use irFAXRecord and wait for an IRE_FAXRECORD_DONE event. Optionally, CNG detection can precede the irFAXRecord by calling irSetParam(IRP_FAX_TONE) to turn CNG detection on and waiting for an IRE_FAX_TONE (before calling irFAXRecord). CNG tone detection can also be turned off by using irSetParam(cid, IRP_FAX_TONE, 0).

The irStop stops the current fax activity and irVCount can be used to obtain the number of FAX bytes recorded.

Fax Resources

All FAX API calls acquire FAX/CNG resources. IRP_RESOURCED_RETURNMODE must be set to IRD_IMMEDIATE. Also, irReserveResource is not supported.

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 message queue key of the calling process if successful. IRR_FAIL is returned if an error occurs. The RM controls send and receive messages fo IRAPI processes.

A process may obtain the 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).

Sending Messages to 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 C, C-Library Functions).

- 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 on a busy system. On a lighly loaded system, an IRAPI timer can go off up to one second after it was requested.

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 6, Message Logger , and Appendix C, C-Library Functions. 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(3IRAPI) and irQTrace(3IRAPI) — Channel level tracing. Messages printed with these functions appear in the system trace if the specified channel is being traced. IrQTrace(3IRAPI) is a macro that executes more quickly than irTrace(3IRAPI), but does not allow a variable number of arguments.

- irTraceP(3IRAPI) and irQTraceP(3IRAPI) — Process level and all error tracing. Messages printed with these functions appear in the system trace if the process printing them is being traced. irQTraceP(3IRAPI) is a macro that executes more quickly than irTrace(3IRAPI), but does not allow a variable number of arguments.

Note: Do not call irTrace(3IRAPI) or irQTrace(3IRAPI) with channel number of -1 (minus one). If this occurs, the trace message will always be displayed regardless of whether the process or channel is being traced.

- irTrace_Put(3IRAPI) — Backward compatibility to **db_put(3SPP)**. This function prints a message to the trace output unconditionally.
- irTRACECHAN_CHK(3IRAPI) and irTRACEPROC_CHK(3IRAPI) — 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(3IRAPI), irSetTraceQkey(3IRAPI), irSetTraceArea(3IRAPI), irSetTraceLevel(3IRAPI), irSetTraceLogMode(3IRAPI), and irSetTraceDateMode(3IRAPI) — 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(3IRAPI) — Character string describing an error
- irErrorName(3IRAPI) — Symbolic name of an error code
- irPError(3IRAPI) — Print to stderr an error description and user text
- irPName(3IRAPI) — Parameter name [see IrPARAMETERS(4IRAPI)]
- irSName(3IRAPI) — Library state name [IrSTATES(4IRAPI)]
- irEName(3IRAPI) — Event name [IrEVENTS(4IRAPI)]
- irEMName(3IRAPI) — Event modifier name [IrEVENTS(4IRAPI)]
- irAName(3IRAPI) — Algorithm name [IrALGORITHMS(4IRAPI)]
- irEFName(3IRAPI) — Event flag name
- irVPName(3IRAPI) — Varparam name
- irGPName(3IRAPI) — String representing a global parameter
- irHName(3IRAPI) — Hardware type name (**/att/include/hwrtype.h**)
- irSvcStName(3IRAPI) — Service state name [IrDEFINES(4IRAPI)]
- irCName(3IRAPI) — Capability name [IrRESOURCES(4IRAPI)]

- irPrintEvent(3IRAPI) — Formatted string of event structure [IrEVENTS(4IRAPI)]
- irPRIcmdName(3IRAPI) — PRI letter command number
- irPRImtName(3IRAPI) — PRI letter message type
- irXName (3IRAPI) — Extension function name (see IrEXTEND (4IRAPI)

Application Dispatch Interface

The IRAPI maintains a default owner for a channel, which 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 in Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration,* 585-313-507, 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

This section discusses the steps necessary to compile and install an Intuity Response Application Programming Interface (IRAPI) application on the system. The compile and install procedure uses the chantest.c application, available on the system in the file **/vs/examples/IRAPI/chantest.c**, as an example.

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

Compiling and Installing 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.

   ```
   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 UCS 1000 R4.2 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 UnixWare **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(1IRAPI)** 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 when the voice system is started and continue running until the voice system is stopped. See **defService** in Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration,* 585-313-507, 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. See the **assign service** command in Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration*, 585-313-507. 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 Applications

The following tools are available to an application developer when debugging an IRAPI application. These tools include both UnixWare and system tools. For more information on the system tools, see Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration*, 585-313-507. For more information on the UnixWare tools, see the *UnixWare Command Reference*.

- **bbs** — UCS 1000 R4.2
- **debug** — UnixWare
- **irevmon** — UCS 1000 R4.2
- **logCat** — UCS 1000 R4.2
- **rmdb** — UCS 1000 R4.2
- **shmview** — UCS 1000 R4.2
- **trace** — UCS 1000 R4.2
- **truss** — UnixWare
- **vtlmgr** — UnixWare

bbs

The **bbs** command can be used to display the voice system bulletin board used for interprocess communication.

debug

The UnixWare **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. See the *UnixWare Command Reference* and the *UnixWare Programming in Standard C and C++* for detailed information on the use of this command.

Note the following caveats 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.

irevmon

The **irevmon** command is a mechanism to monitor events for processes and/or channels.

logCat

The **logCat** command is 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**.

rmdb

The **rmdb** command is used to display internal tables maintained by the Resource Manager (RM) and to control the verbosity of RM trace messages. See Using rmdb to Assess Resource Utilization on page 329, and Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration,* 585-313-507, for additional information on **rmdb**.

shmview

The **shmview** command is used to display internal shared memory segments maintained by the IRAPI and run time systems.

trace

The **trace(1IRAPI)** command is used to get detailed information about the program flow for IRAPI applications and other system 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 UnixWare **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. The *lbolt* option includes the system lbolt in the trace output. The *lbolt* option is a counter that starts with 0 when the system is booted and incremented every 1/100th of a second.

trace tsm date lbolt 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 above).

As described in IRAPI Run-Time Services on page 216, 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.

truss

The UnixWare **truss(1)** command can be used to display all system calls made by a process. The process may already be running, in which case **truss** attaches to the process image. You can also use the **truss(1)** command to start a process by using the process name as an argument to the **truss(1)** command.

vtlmgr

Note: This tool is only available from the system console.

If better X-windowing tools are not available, it is often useful to use the UnixWare **vtlmgr(1)** command to have multiple terminal sessions available.

Performance and System Tuning for IRAPI Applications

This section 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.
- When a resource card is brought in service, it is assigned a saturated usage. The saturated usage of the card can vary depending on the physical configuration. Users can examine the usage values of resource cards using the **rmdb** command (see Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration,* 585-313-507).
- 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: You can also examine pending requests using the **rmdb** command. See Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration*, 585-313-507, 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 -c** command shows:

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

```
rmdb -c 14
Card table:
14  :  MTC_INSERV     2           0x00000000
         0  "SP_RECORD                  " [83      0      0      0    ] [ 0/0 /120]
         1  "SP_ECHOCAN                 " [150     0      0      0    ] [ 0/0 /60]
          2  "SP_PLAY                    " [83      0      0      0    ] [ 0/0 /120]
          3  "SP_TTS                      " [166     0      0      0    ] [ 0/0 /60]
         4  "SP_WW_RECOG                " [600     0      0      0    ] [ 0/0 /15]
         5  "SP_CELPCODE                " [166     0      0      0    ] [ 0/0 /60]
         Saturated     : [10000  10000   10000   10000 ]
         High Water    : [0      0       0       0     ]
         Current       : [0      0       0       0     ]
         Allocation #  : 0x0       0x0       0x0       0x0
rmdb -c 12
12  :  MTC_INSERV     0           0x00000000
```

```
0  "LSPS_RECORD    " [78       200     0       125   ] [ 0/0 /50]
1  "LSPS_PLAY      " [78       0       200     125   ] [ 1/1 /50]
2  "LSPS_WW_RECOG  " [333      200     0       0     ] [ 0/0 /30]
3  "LSPS_DATA_COLLECT" [2500   500     0       0     ] [ 0/0 /1 ]
4  "LSPS_BUS_DIAG  " [1250     1500    1500    1500  ] [ 0/0 /1 ]
Saturated     : [10000  10000   10000   10000 ]
High Water    : [78     0       200     125   ]
Current       : [78     0       200     125   ]
Allocation #  : 0x0       0x1       0x0       0xn,
```

Disk Performance

One of the major performance bottlenecks in the 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, two 8K blocks are allocated for it. The remaining 7K 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 i486    11/27/96
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 an 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 Tunable Parameters

The RM tunables are listed below. The default values for each parameter are shown in Table 21 on page 365, 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 372 bytes. The amount of space devoted to the channel table inside RM is 45,012 bytes.

Note: These parameters are set to support most configurations of the system. Ordinarily, it should not be necessary for these parameters to be tuned. However, tuning may be necessary in some particularly challenging configurations.

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 remote maintenance board (RMB), etc., do not require entries in the card table. This parameter normally should not have to be tuned.

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.

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 system depend on allocating an appropriate number of channel groups.

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

NCONTEXTSTACK

The NCONTEXTSTACK parameter specifies the maximum depth of the context stack for subprograms. This parameter should only require tuning on systems that run applications that include subprograms that call other subprograms.

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

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

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

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.

NTDM

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

The following table summarizes each of the RM tunable parameters, its default, and its size are shown in Table 21 on page 365.

Table 21. Resource Manager Tunable Parameters

Parameter Description	Parameter Name	Default	Size
Number of network interface, resource cards	NCARDS	25	604
Number of channels	NCHANNELS	145(if cPCI) 228 (if ISA)	216
Number of channel groups	NCHANNELGROUPS	32	24
Size of channel's context stack	NCONTEXTSTACK	3	1
Number of processes that can simultaneously open	NDEVICES	64	72
Number of callout entries (for channel or process timeouts)	NCALLOUTS	2000	24
Maximum message size	MAXMSGSZ	2048	2048
Maximum messages in message queue	MAXMSGQl	500	varies (up to MAXMSGSZ)

1 of 2

Table 21. Resource Manager Tunable Parameters

Parameter Description	Parameter Name	Default	Size
Number of dynamic resource structures	NDYNSTRUCT	432	24
Number of functions supported by all packs	NFUNCTIONS	32	88
Number of TDM busses	NTDM	1	33,772
Size of the channel's profile	PROFILE_SIZE	3300	33
Number of Qkeys	NQKEYS	215	28

2 of 2

Parameter Tuning Procedure

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

1. Enter **/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.

2. Enter **/etc/conf/bin/idbuild -B**

3. Reboot the system.

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 the online instructions on the system about irAPI.rc(4IRAPI) 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.

6 Message Logger

Overview

This chapter describes:

- The message logger environment
- The procedure to use the message logger with customer-defined data interface processes (DIPs)
- The procedure to add and change system message explain text

The purpose of this chapter is to provide application developers with the information required to use the message logger to create and change error messages and explain text for an application or DIP.

Overview of the Message Logger

The message logger provides facilities that allow UnixWare processes to log messages to predefined destinations. Messages are logged from processes coded as C-language programs, for example, custom DIPs.

Note: Refer to the **msgadm** command in *UCS 1000 R4.2 Administration*, 585-313-507.

Message Logger Purpose

Logger messages are typically used to alert administrators or operators of errors encountered as calls are processed. For example, a DIP can report an error to the message logger if it received data that could not be processed. Logger messages can also be used to inform administrators and operators of events completed by the calling process. For example, a DIP can report to the message logger that it has successfully started and initiated a process.

Message Classes

Messages are categorized into different “classes” depending on feature package, subprocess, or module that logged them.

Some examples of message classes are:

- MTC — logged by the maintenance module of IRAPI
- ASAI — logged by the Adjunct/Switch Application Interface feature package
- TSM — logged by the transaction state machine

For more information about message classes, see the *UCS 1000 R4.2 System Reference*, 585-313-210.

Message Logger Development

Several steps are involved in developing messages in custom software. After determining points in the source code where messages should be logged, you must specify the exact structure and wording of the message. This includes specifying what text should be *hard coded* in the messages and what text should be *variable*, that is, provided by the process at run time. Variable text can include such things as an error code, channel number, or a reason string, and can be a character string or integer.

Message Logger Structure

All messages in the logger system are organized internally by an indexing scheme.

Format Text

Message format text is in the **{*CLASS*}msg** files found in the **/vs/spool/log/formats** directory. Custom DIPs must use the **APPLmsg** format file.

CAUTION:
Do not modify or add to any messages files other than ***APPLmsg*** files. Doing so could render the entire message logger system inoperable or produce unintelligible messages in the log files.

Header Files

The message header files are found **/vs/spool/log/head** and provide the internal index of the message to the DIP or C-language program that originally sent the message. There must be a definition in the **logAPPL.h** file for each message used from the **APPLmsg** file. These definitions must be sequential starting at 1 and the index in **logAPPL.h** list must match the *msgID* in **APPLmsg** file.

CAUTION:
Do not modify or add to any message header files other than ***logAPPL.h*** header files.

DIPs can, however, call message classes already defined in header files other than **logAPPL.h**, if applicable.

The example provided in logAPPL.h File on page 402, in Appendix A, Application Example shows the following three message definitions in `logAPPL.h`:

```
#define APPL_INITFAILlogAPPL(1)
#define APPL_MSGSNDERRlogAPPL(2)
#define APPL_UNKNOWNMSGlogAPPL(3)
```

These correlate to three messages found in the **APPLmsg** file, APPL001, APPL002, and APPL003, also shown in “logAPPL.h File”.

Rules Files

Each message must also have an associated rules file. The rules file is an ASCII text file and is owned by the message class (such as, maintenance – MTC).

Message Content and Format Specification

Message content and format is specified in the **APPLmsg** file found in **/vs/spool/log/formats**. The following rules govern the parsing of this file:

- All blank lines are ignored.
- Comment lines are those with a "+", character in column 1.
- Each non-blank, non-commented line allocates one message.
- A message can span multiple lines using the “\” character at the end of the line to indicate continuation to the next line.

While you can specify tab (“\t”) and newline (“\n”) characters in the message text, it is not recommended since output message text formatting is handled by the display messages command.

The standard UCS 1000 R4.2 message text appears as follows:

{msgID} {FRU} {EQ} {EQ#} ({MNEMONIC}) {Message Text}

In the case of **APPLmsg**, *{msgID}* is in the form **APPL*NNN*** where *NNN* is a number ranging from 001 to the number of messages in the class. *NNN* must match the index for the message it is specifying in the **logAPPL.h** file. *{msgID}* must occupy eight spaces; if *{msgID}* is less than eight spaces it must be right-filled with blanks. The **APPLmsg** file is delivered with placeholders for messages APPL001 through APPL032. More can be added if needed. To remove message IDs from the system, you must to replace the error message with a placeholder or a new error message.

{FRU} is a two-character field indicating a field-replaceable unit. Examples in the UCS 1000 R4.2 environment include TR, T1, SP. If it is not necessary to specify a field-replaceable unit, use a double dash as the value in this field.

{EQ} is a two-character field indicating the type of resource to which the message applies, for example, CH for channel, CA for card, or a double dash, if not applicable.

{EQ#} is a three-digit number to specify the particular card or channel, for example, CH1 for channel 1, CA1 for card 1, or a triple dash, if not applicable).

({MNEMONIC}) is the #define symbol used in the **logAPPL.h** file to define the message. Note that the MNEMONIC has parenthesis around it when the field is specified in **APPLmsg**.

{Message Text} is the text associated with the message. The message text can span multiple lines by placing the backslash (\) at the end of the line.

Message Parameters

Parameters are the variable text items provided by the calling process at run time. There are two types of parameters: character string, denoted with %s, and integer, denoted with %d. These parameters may appear anywhere within the *{Message Text}* area.

Most DIPs do not use the *{FRU}, {EQ},* and *{EQ#}* fields. Therefore, these fields can usually be specified as --, --, and ---, respectively.

If you do use the *{FRU}*, *{EQ}*, and *{EQ#}* as parameters in the **APPLmsg** file, you must specify them as follows:

```
{FRU}%.2s
{EQ}%.2s
{EQ#}%.3d
```

Messages defined for the UCS 1000 R4.2 included an optional *<<{NAME},{TYPE}>>* field following each parameter specification. This field is reserved for future use, and may be omitted from messages defined in the **APPLmsg** file.

Message Mnemonic Definition

You must modify **logAPPL.h** header file, located in **/vs/spool/log/head,** to include any new messages specified in the **APPLmsg** file. Insert message definitions in the file on the line preceding the last #endif statement. See logAPPL.h File on page 402 in Appendix A, Application Example, for an example of where to place the new messages.

In general, **logAPPL.h** entries should appear as follows:

#define *{MNEMONIC}* logAPPL(*{N}*)

where *{MNEMONIC}* is a define symbol for the message. By convention, the form of the mnemonic is *{CLASS}_{NAME}* where *{CLASS}* is APPL and *{NAME}* is a descriptive word or abbreviation for the message. *{N}* is the message number within the APPL class of messages. It is important that **logAPPL(1)** correspond to the message defined for APPL001, **logAPPL(2)** correspond to the message defined for APPL002, etc.

Note: Many unused message IDs are allocated in the **APPLmsg** file. A corresponding mnemonic is not required for unused messages.

Message Rules File Definition

Each module's rules file is located in its own *registration* directory. For example, the rules file of the maintenance module is in **/mtce/registration/rules**.

Compiling the Messages in the DIP

Note: The following procedure assumes that the **APPLmsg** file already contains the message text and the logAPPL.h already contains the mnemonics.

Use the following procedure to compile new messages in associated with a DIP:

1. Include the following logger header files in the DIP code.

```
#include "/vs/spool/log/head/log.h"
#include "/vs/spool/log/head/systemLog.h"
#include "/vs/spool/log/head/logAPPL.h"
```

2. Place a call to **logInit(3x)** within the DIP to initialize the DIP/logger interface. **logInit(3x)** has the format **logInit (program_name)** where *program_name* is normally an all-upper-case representation of the name of the executable (for example, "MYDIP").

   See logMsg on page 646 in Appendix C, C-Library Functions, for additional information on **logInit**.

3. Place calls to **logMsg(3x)** within the DIP to send specified messages to the logger. logMsg has the format

   logMsg (MNEMONIC,EL_FL,arg1,arg2, . . .)

   where *MNEMONIC* is the message mnemonic for the system message, *EL_FL* is a macro that identifies the file name and line number in the code

where the call was generated, and the *arg1,arg2,...* are the parameters to the message text.

Note: By default, the message mnemonic does not appear in output from the **display messages** command. Use the **logCat** command to display messages and their corresponding mnemonics. Use the **logFmt** command to enable and disable the appearance of the message mnemonic. See Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration,* 585-313-507, for additional information about **logCat**.

See logMsg on page 646 in Appendix C, C-Library Functions, for additional information on **logMsg**.

4 Whenever **logAPPL.h** is changed, the error message IDs are not known to the DIP until run time. To make the message IDs known to the DIP, you must first add **-D _INSTALLABLE_APPL** in the DIP compilation statement, as shown in Chapter 4, Data Interface Processes . Secondly, add the **Fcn_APPLMSG_START** function to the DIP code, as shown in the following example:

```
#include <stdio.h>
#include <sys/types.h>

#include “/vs/spool/log/head/log.h”

int Fcn_APPLMSG_START()
```

```
   {
      static int startLoc = -1;
      static int readID;

/*    Have the message classes been read again? If
/*    /*    so, make sure we get the new value
*/

      if (logClassReadCnt  != readID)
         {
         readID = logClassReadCnt ;
         startLoc = -1 ;
         }

      if (startLoc < 0)
         startLoc = logStartClass("APPL") ;
      return(startLoc) ;

   }
```

5 Rebuild the logger format and message indexing files and reinitialize the logger by executing the following commands at the system prompt:

```
cd /vs/spool/log/formats
make -f formats.mk install
reinitLog
```

Note: If **reinitLog** fails, execute **touch /vs/spool/log/head/***, then try **reinitLog** again.

See Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration,* 585-313-507, for additional information about **reinitLog**.

6 Compile the DIP code by linking the logger library files found in **/vs/lib/liblog.a** and **/vs/lib/libprism.a**. See Step 8: Compile and Execute the DIP on page 172 in Chapter 4, Data Interface Processes for details on compiling and executing the DIP.

7 Modify the new logger message priority, destination, and threshold using the system message administration procedures provided in Chapter 3, “Voice System Administration,” in *UCS 1000 R4.2 Administration,* 585-313-5017.

8 Test your messages as described in the following sections.

Testing a Single Error Message

Use the following example as a guideline to test a single error message. This example shows how to test the sample message APPL001 (APPL_INITFAIL) in Sample DIP on page 393 in Appendix A, Application Example. View the **APPLmsg** file to see what strings are necessary to fill in the error, and the view the **logAPPL.h** file for the index of the error, for example, logAPPL(1).

1. Enter **cd /vs/spool/log/formats**
2. Enter the following with a carriage return after each line:

   logTest
   0 0x01 2 stock_dip logAPPL(1) stock_dip “Could not open stocks file”
   <del>

   This logs one occurrence of the APPL001 error in the log.
3. To verify the error is correct, enter **logCat -t3** to see the last three error messages. Your message should be among the messages displayed. For example:

```
*  Wed Feb 23 12:30:53 1998 stock_dip logTest.c:418
APPL001 -- -- -- Application DIP ‘stock_dip’ failed to
initialize. Reason: Could not open stocks file.
```

Testing Several Error Messages

To test several error messages at the same time, you can create a file with the inputs to **logTest** for all the messages. This is an easy way to test all the messages your DIP uses. The following example shows how to test the sample messages APPL001, APPL002, and APPL003.

1. Create a file called **/tmp/appl_errors** that contains the following:

   0 0x01 2 stock_dip logAPPL(1) stock_dip "Could not open stock file"
   1 0x01 2 stock_dip logAPPL(2) stock_dip
   1 0x01 2 stock_dip logAPPL(3) stock_dip

2. Enter the following:

   cd /vs/spool/log/formats
   logTest < /tmp/appl_errors

3. Check the error log to make sure your messages are correct by entering **logCat -t3**

   The system displays a message similar to the following:

```
*  Wed Feb 23 12:39:56 1998 stock_dip logTest.c:418
APPL001 -- -- -- Application DIP 'stock_dip' failed to
initialize. Reason: Could not open stock file.
*  Wed Feb 23 12:39:57 1998 stock_dip logTest.c:418
APPL002 -- -- --- Application DIP 'stock_dip' failed to send
message to script.
```

```
*  Wed Feb 23 12:39:58 1998 stock_dip logTest.c:418
APPL003 -- -- --- Application DIP 'stock_dip' received an
unknown message.
```

Note: If any messages are incorrect (that is, the data provided does not match the error format), the logger prints an expansion failure error.

For more information about **logTest** and **logCat**, see Appendix A, "Summary of Commands," in *UCS 1000 R4.2 Administration*, 585-313-507.

Adding and Changing Explain Message Text

Use the text editor or the command line to add and/or change explain message text.

Using the Text Editor to Add Messages

Use the following procedure to add new explain message text using the text editor:

1. Create the explanation text file in the **/gendb/data/explain/*<alphabetic_letter>*** directory, where *alphabetical_letter* matches the first letter of the explain message. That is, AL_RESET_STA must appear in the A directory, TSM_NOSER must appear in the T directory, etc.

2 Add the name of the explain text file and any aliases to the **/gendb/data/explain/translateLst** file, as shown in the examples below

name	Directory	12-Character Alias
ALERT002	AL_RESET_STATS	AL_RESET_STA
ALERT003	AL_INVALID_THRESHOLD	AL_INVALID_T
ASAI002	A_DSCRIPT_TERM	A_DSCRIPT_TE

3 Enter **:wq!** to save the information and exit the file.

Using the Command Line To Add Messages

Use the following procedure to add new explain message text using the **edExplain** command:

1 Log in to the system as root.

2 At the system prompt, enter **edExplain <*msg_id*>** where *<msg_id>* is the messages ID (for example, APPL001).

See the **edExplain** command in Appendix A, “Summary of Commands,” in *UCS 1000 R4.2 Administration,* 585-313-507.

Removing Error Messages

Use the following procedure to remove error messages:

1. Edit **/vs/spool/log/head/logAPPL.h** and remove the *#define* statements for the custom error messages.

   For example, in logAPPL.h File on page 402 in Appendix A, Application Example, remove the following statements:

   ```
   #define APPL_INITFAIL logAPPL(1)
   #define APPL_MSGSNDERR logAPPL(2)
   #define APPL_UNKNOWNMSG logAPPL(3)
   ```

2. Edit **/vs/spool/log/formats/APPLmsg** and replace the custom error definitions with placeholders.

   For example, in APPLmsg File on page 400 in Appendix A, Application Example, replace the following statements:

   ```
   APPL001  -- -- -- (APPL_IN ITFAIL) Application DIP ‘%s’
   failed to initialize. Reason: %s.
   APPL002  -- -- --- (APPL_MSGSNDERR) Application DIP ‘%s’
   failed to send message to script.
   APPL003  -- -- --- (APPL_UNKNOWNMSG) Application DIP ‘%s’
   received an unknown message.
   ```

with:

```
APPL001  -- -- --- ({MNEMONIC}) %s
APPL002  -- -- --- ({MNEMONIC}) %s
APPL003  -- -- --- ({MNEMONIC}) %s
```

3 Rebuild the errors file by entering the following:

```
cd /vs/spool/log/formats
make -f formats.mk install
reinitLog
```

The system removes your custom error messages from the log.

A Application Example

Overview

This appendix presents an example of an application. It includes:

- The TAS script instructions
- A data interface process (DIP)
- An external function, diptest, that calls the DIP

This application is a very simple example of a banking application. The script prompts the caller for a social security number and a six-digit account number. The social security number and account number are passed to the DIP via the dbase instruction within the external function. In a real-life application, the DIP probably would use the social security and account numbers to access a local or remote database to retrieve information about that account.

For simplicity, the sample DIP simply manipulates the social security number and returns the result (last four digits of social security number) as the account balance to the script. After the account balance is spoken to the caller, the caller is given the chance to enter another social security number

and account number or to quit. You can use this sample application as a model in building other applications for the system.

This appendix also provides a sample fax print and record example.

Sample Script — TAS Script Language

The following example shows the same script in the TAS script language described in Chapter 3, TAS Script Instructions .

```
/* TSM script application dipscript */

#include "dipscript.h"
    tfile("std_speech.pl" "dipscript.pl")
    event(0,L__quit)
    event(1,L__quit)
L_start:
    trace(im.1)
    tic    ('a')
    trace  (im.2)
    talk   ("Hello, this is the DIP test script" )
L_entry_loop:
    load   (int.F_acct_balance, im.0)
    trace  (im.3, int.F_acct_balance)
    /* get caller input */
    ttdelim(-1, -1, -1, -1)
    load   (int.F__CI_TRIES_USED, im.0)
```

```
L_4:          /* prompt */
   talk   ("Please enter your SSN" )
L_5:         /* try again */
   tttime (5, 5)
   incr   (int.F__CI_TRIES_USED, im.1)
   getdig (0, ch.F_ssn, im.09)
   load   (int.F__CI_NO_DIGS_GOT, r.0)/* save for user */
   trace  (im.4, ch.F_ssn)
   jmp    (r.0 == im.0, L_2)
   jmp    (int.F__CI_NO_DIGS_GOT < im.09, L_3)
   goto   (L_1)
L_2:  /* Initial timeout */
   jmp    (int.F__CI_TRIES_USED == im.3, L__quit)
   goto(L_4)
L_3: /* too few digits */
   jmp    (int.F__CI_TRIES_USED == im.3, L__quit)
   goto   (L_4)
L_1:
   /* get caller input */
   ttdelim(-1, -1, -1, -1)
   load   (int.F__CI_TRIES_USED, im.0)

L_9: /* prompt */
   talk   ("Please enter your account number" )
L_10: /* try again */
   tttime (5, 5)
   incr   (int.F__CI_TRIES_USED, im.1)
   getdig (0, ch.F_acct_num, im.06)
```

```
    load   (int.F__CI_NO_DIGS_GOT, r.0) /* save for
           user */
    trace  (im.5, ch.F_acct_num)
    jmp    (r.0 == im.0, L_7)
    jmp    (int.F__CI_NO_DIGS_GOT < im.06, L_8)
    goto   (L_6)
L_7: /* Initial timeout */
    jmp    (int.F__CI_TRIES_USED == im.3, L__quit)
    goto   (L_9)
L_8: /* too few digits */
    jmp    (int.F__CI_TRIES_USED == im.3, L__quit)
    goto   (L_9)
L_6:
    trace  (im.6,ch.F_ssn)
    trace  (im.6,ch.F_acct_num)
    L__diptest(im.F_ssn, im.F_acct_num)
    load   (int.F_acct_balance, r.0)
    trace  (im.6, r.0)
    trace  (im.7)
    jmp    (int.F_acct_balance >= im.0, L_11)
    trace  (im.8)
    talk   ("This is an error situation, the return
        value is" )
    tnum   (int.F_acct_balance, 't')
    trace  (im.9)
    goto   (L__quit)
L_11:
    load   (int.F_acct_balance, int.F_acct_balance)
```

```
    trace  (im.10, int.F_acct_balance)
    trace  (im.11)
    talk   ("Your account balance is" )
    L__sp_dol(int.F_acct_balance)
            /* get caller input */
    ttdelim(-1, -1, -1, -1)
    load   (int.F__CI_TRIES_USED, im.0)
L_15: /* prompt */
    talk   ("Enter 1 to enter ssn again, 2 to quit" )
L_16:  /* try again */
    tttime (5, 5)
    incr   (int.F__CI_TRIES_USED, im.1)
    getdig (0, ch.F_prompt, im.01)
    load   (int.F__CI_NO_DIGS_GOT, r.0) /* save for
        user */
    trace  (im.12, ch.F_prompt)
    jmp    (r.0 == im.0, L_13)
    jmp    (ch.F_prompt == im.'1', L_entry_loop)
    jmp    (ch.F_prompt == im.'2', L_12)
    goto   (L_12)
L_13: /* Initial timeout */
    jmp    (int.F__CI_TRIES_USED == im.3, L__quit)
    goto   (L_15)
L_12:
    trace  (im.13)
    talk   ("Thank you for calling the Dip Test
        Script" )
    trace  (im.14)
```

```
   tic    ('h')
   trace  (im.15)
   goto   (L__quit)


L__quit:
   quit()
L__save_events:
   rts()

L__seasonal_greetings: /* play seasons greetings messages,
        if any             */
   rts()
#include "/vs/bin/ag/lib/_sp_dol.t"
#include "diptest.t"
```

Sample External Function

The following is an example of an external function. This external function is called by the script included in this chapter. In this example, the external function is in the same directory as the script.

```
/*
* FUNCTION diptest – DIP test script sample external
* function
* INPUTS:
*      SSN – field with the SSN
*      acct num – field with the account number
* RETURNS: account balance >= 0 , failure < 0
*/


DEFARG_COUNT(2)
DEFARG(ssn,char,in) /* r.3 */
DEFARG(acct_num,char,in) /* r.2 */


#define ACCT_REQ 8500 /* mcont for the DIP */
#define SZUV17 /* length of ssn and acct num */
#define F__TEMP10 10 /* 10-byte offset into TEMP */


L__diptest:
    strcpy(ch.F__TEMP, *ch.3) /* load ssn number */
    strcpy(ch.F__TEMP10, *ch.2) /* load acct number */
```

```
dbase( im."bankMgrDip", ACCT_REQ, ch.F__TEMP, im.5,
    ch.F__TEMP, SZUV )
trace (im.18001, r.0) /* mcont is returned into r.0 */
load ( r.0, int.F__TEMP)
trace (im.18006, r.0)
rts()
```

Sample DIP

The following is an example of a DIP. This is the same example used in Chapter 4, Data Interface Processes , except that the DIP has been modified to manipulate the social security number as described in the Overview on page 386.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/mesg.h>
#include  "/usr/spool/log/head/log.h"
#include  "/usr/spool/log/head/systemLog.h"
#include  "/usr/spool/log/head/logAPPL.h"
#include  "shmemtab.h"
#include  "spp.h"
#include  "mesg.h"
#include  "VS.h"
```

```
/* Define all messages that can be received

* For example, caller_info_msg is a message that is sent by
* the TSM script giving the caller's social security number.
* Also define the message ids for each messages received.
* These message ids should be in a header file instead of
* here.
*/

#define ACCOUNT_REQUEST 8500
struct callerMsg {
   struct mbhdr  hd;
   char socialSecurityNo[10];
   char accountNo[7];
};

/* Define Message Receive structure
* Should be large enough to hold largest message.
* Add all received message structures in the following
* union.
*/

union rcvMsg {
   struct ms_univ u; /* the standard message (mesg.h) */
   struct callerMsg   c; /* caller's info */
};
```

```
/* Define Message structures to be Sent.
* Also define the message id for each message.
* The message id should go in a header file but
*  it's shown here for convenience.
* The message ids should all be unique across all
* applications.
* Only one message is sent in this example but usually you'll
*  lots more.
*/


#define ACCOUNT_INFO 8090 /* message id */
struct accountMsg {
   struct mbhdr  hd;
   int           balance;
};

static char   *Myname="bankMgrDip"; /* Name of this DIP */
static short  Myinstance=1;       /* Instance of DIP */


/* Names of other processes you talk to */
#define DBDIPPER "bankTellerDip"

main()
{
   int                 myQkey;
   int                 noBytesRead;
   int                 accountBalance;
   int                 retCode;
```

```
union rcvMsg        rcvbuf;
struct accountMsg   acctbuf;


/* initialize DIP */


/* Logger Initialization */

/* Sleep if necessary to wait for the voice system
/* to finish diagnosing the cards */

logInit(Myname);

/* Get your dynamically-assigned Qkey */
myQkey = VSstartup(Myname, Myinstance, DIP_PROC);
if (myQkey <= 0 ) {
    db_pr("%s: Can't get qkey: VSstartup: %s\n");
        VSerror(myQkey);
    logMsg(APPL_INITFAIL,EL_FL,*Myname,"Can't get

        qkey");
    sleep(5); /* to slow down continuous respawning */
    exit(1);
}

/* Clear out my message queue */
noBytesRead = mesgrcv(myQkey, &rcvbuf, sizeof(rcvbuf),
       0, IPC_NOWAIT,NULL);
while (noBytesRead >= 0) {
noBytesRead = mesgrcv(myQkey, &rcvbuf, sizeof(rcvbuf),
```

```
        0, IPC_NOWAIT,NULL);
}

/* Main processing Loop:
* Read and process message for ever
*/
while (1) {
    /* wait for a message to arrive */
    noBytesRead = mesgrcv(myQkey, &rcvbuf, sizeof(rcvbuf),
        0, 0);

if (noBytesRead < 0) {
    /* Something went wrong with the read
    * Could be that the reading was interrupted (EINTR).
    * There should be some error processing here but
    *       for brevity I'll just try to read again.
    */
    continue;
    }

/* Got a message! Get to work */
db_pr("%s: got message: chan =%d, id=%d, senderQkey=%d\n",
    *Myname,rcvbuf.c.hd.mchan, rcvbuf.c.hd.mcont,
    rcvbuf.c.hd.morig);

/* Typically, the DIP will have a case for each
*      possible message id.
* In this example, we only have one possible message
```

```
*      that can be received.
*/
switch (rcvbuf.c.hd.mcont) {
case ACCOUNT_REQUEST:
   /* TSM script wants account balance info */
   db_pr("%s: request for account info for SS#=%s\n",
      *Myname, rcvbuf.c.socialSecurityNo);
   db_pr("%s:    and account#=%s\n",
      rcvbuf.c.accountNo);
   /* Go out and get the account information
   * and return it in accountBalance.
   * This balance (for simplicity) is generated from
   * the last 4 digits of the SSN
   */
   accountBalance = atoi(rcvbuf.c.socialSecurityNo+5);
   db_pr("%s:    the balance = %d\n",, *Myname,
      accountBalance );

   /* Now package and send respond back */
   acctbuf.hd.mchan = rcvbuf.c.hd.mchan;
   acctbuf.hd.mtype = 1;
   acctbuf.hd.morig = myQkey;
   acctbuf.hd.mcont = ACCOUNT_INFO;
   acctbuf.hd.mseqno = 0;
   acctbuf.balance = accountBalance;
   retCode = mesgsnd(TSM, &acctbuf, sizeof(acctbuf), 0);
   if (retCode < 0) {
      /* Message send failed; log message
```

```
            * Note that before this will work you
            * must add your DIP errors into the logger system.
            */
            logMsg(APPL_MSGSNDERR, EL_FL, acctbuf.hd.mchan,
               *Myname);
         }
         break;
      default:
         /* Notify logget that an unknown message was
         *   received.
         */
         logMsg(APPL_UNKNOWNMSG, EL_FL, rcvbuf.c.hd.mchan,
            *Myname );
         break;
      } /* switch on message id */
        } /* while loop that reads forever */
} /* Main program */

int Fcn_APPLMSG_START()

{
   static int startLoc = -1 ;
   static int readID ;

/*  Have the message classes been read again? If so,  */
/*  make sure we get the new value                                */

   if (logClassReadCnt  != readID)
```

```
        {
        readID = logClassReadCnt ;
        startLoc = -1 ;
        }
    if (startLoc < 0)
        startLoc = logStartClass(“APPL”) ;
    return(startLoc) ;
}
```

APPLmsg File

The following messages are samples in the **/usr/spool/log/formats/APPLmsg** file to be modified as required for your application. *DO NOT* add new messages unless all of the messages in this file have already been used. If you must extend the file, add a block of unused messages, so that you do not extend the file each time you add one new message.

```
APPL001 -- -- --- (APPL_INITFAIL)\
Application DIP ’%s’ failed to initialize. \
Reason: %s.

APPL002 -- -- %3d (APPL_MSGSNDERR) \
Application DIP ’%s’ failed to send message to script.
```

```
APPL003 -- -- %3d (APPL_UNKNOWNMSG) \
Application DIP '%s' received an unknown message.

APPL004 -- -- --- ({MNEMONIC}) %s
APPL005 -- -- --- ({MNEMONIC}) %s
APPL006 -- -- --- ({MNEMONIC}) %s
APPL007 -- -- --- ({MNEMONIC}) %s
APPL008 -- -- --- ({MNEMONIC}) %s
APPL009 -- -- --- ({MNEMONIC}) %s
APPL010 -- -- --- ({MNEMONIC}) %s
APPL011 -- -- --- ({MNEMONIC}) %s
APPL012 -- -- --- ({MNEMONIC}) %s
APPL013 -- -- --- ({MNEMONIC}) %s
APPL014 -- -- --- ({MNEMONIC}) %s
APPL015 -- -- --- ({MNEMONIC}) %s
APPL016 -- -- --- ({MNEMONIC}) %s
APPL017 -- -- --- ({MNEMONIC}) %s
APPL018 -- -- --- ({MNEMONIC}) %s
APPL019 -- -- --- ({MNEMONIC}) %s
APPL020 -- -- --- ({MNEMONIC}) %s
APPL021 -- -- --- ({MNEMONIC}) %s
APPL022 -- -- --- ({MNEMONIC}) %s
APPL023 -- -- --- ({MNEMONIC}) %s
APPL024 -- -- --- ({MNEMONIC}) %s
APPL025 -- -- --- ({MNEMONIC}) %s
APPL026 -- -- --- ({MNEMONIC}) %s
APPL027 -- -- --- ({MNEMONIC}) %s
APPL028 -- -- --- ({MNEMONIC}) %s
```

```
APPL029 -- -- --- ({MNEMONIC}) %s
APPL030 -- -- --- ({MNEMONIC}) %s
APPL031 -- -- --- ({MNEMONIC}) %s
APPL032 -- -- --- ({MNEMONIC}) %s
```

logAPPL.h File

The following is the code for the **logAPPL.h** file located in **/usr/spool/log/head**. There must be an definition in **logAPPL.h** for each message used from **APPLmsg**.

```
/* @(#)logAPPL.h 8.1.1.2 16:54:41 6/28/93*/
#ifndef header_LOGAPPL_H
#defineheader_LOGAPPL_H

/* For compatibility with the old and new C++ define:*/

#ifdef    c_plusplus
#ifndef   __CCPLUSPLUS__
#define   __CCPLUSPLUS__
#endif
#endif

#ifdef    __cplusplus
#ifndef   __CCPLUSPLUS__
```

```
#define   __CCPLUSPLUS__
#endif
#ifndef   CC_TYPE_SAFE
#define   CC_TYPE_SAFE
#endif
#endif

#ifdef    _INSTALLABLE_APPL
#ifdef__CCPLUSPLUS__
CC_EXTERN( int Fcn_APPLMSG_START() ; )
inline int logAPPL(int xx)  {return
(Fcn_APPLMSG_START()+(xx)-1); }
#else

extern int Fcn_APPLMSG_START() ;

#define logAPPL(xx) (Fcn_APPLMSG_START()+(xx)-1)
#endif

#else

#ifdef __CCPLUSPLUS__
inline int logAPPL(int xx){return (_APPLMSG_START+(xx)-1); }
#else
#define logAPPL(xx)    (_APPLMSG_START+(xx)-1)
#endif
```

```
#endif

#define APPL_INITFAIL   logAPPL(1)
#define APPL_MSGSNDERR  logAPPL(2)
#define APPL_UNKNOWNMSG logAPPL(3)

#endif
```

IRAPI Script With Speech Recognition

This sample IRAPI script executes the yes/no recognition grammar using the LSPS II circuit card.

```
#include <stdio.h>
#include "irapi.h"
#include <unistd.h>
#include "shmemtab.h"
#include "s51.h"
#include "lspsipGlobals.h"
#include "busDiag.h"
#include "mtc_cmd.h"
#include "lspsmsg.h"
#include "lsps_asr.h"
#include "irlsps.h"
#include "recog_dip.h"

#define T_LVL   IRD_TRACE_LEVEL_16
```

```
#define T_AREA  IRD_TRACE_AREA_16

void myError(char * string);
void myQTrace(char * string);
int iri_tdmlisten(int chan);
int iri_tdmtalk(int chan);
void speak_return(char * text, int chan, ir_key_t qkey);
char * basename(char * str);

union recv_data {
 struct recv_script script;
 struct VARLETTER(MAX_LEN) letrptr[1];
 struct recv_dcdip dcdip;
 };

union send_data {
 struct send_tsm tsm;
 struct send_dcdip dcdip;
 };

struct recv_msg {
 struct mbhdr mhdr;
 union recv_data data;
};

struct send_msg {
 struct mbhdr mhdr;
 union send_data data;
};
```

```
/*
* The following structure helps in sending a message back to TSM
*  only after all secondary choices have been obtained and
processed.
*/

struct wait_struct {
 struct recv_msg recv_mesg;
 int choice_wait_flag[MAX_NUM_CHOICES];
 struct choice choices[MAX_NUM_CHOICES];
 int num_of_frames; /* number of frames used in recognition */
 int time_stamp;
 };

/*
* Structure used for servicing the requests from dcdip
*/

intCHAN;
channel_id cid;

struct recv_script recv_script;
struct recv_msg recv_msg;
struct send_msg * send_msg;

main(int argc, char * argv[])
{
 ir_key_t myKey, qkey, recog_qkey;
```

```
  ir_event_t ev;
  union LSPSIPmsg * dataptr;
  struct mbhdr * hdptr;
  int timeslot, chan, rc, value;
  struct LSPSasr asr;
  struct LSPSstarttimer stimer;
  struct LSPSplay play;
  struct asrStat * stat;
  FILE * fp;
  int PLAY = 0;
  char * baseptr;
  char buf[21];

  baseptr = basename(argv[0]);
  if((myKey = irRegister(baseptr)) == IRR_FAIL)  {
    myError("irRegister failed.  EXITING");
    exit(-1);
  }
  irTrace(CHAN, T_LVL, T_AREA, "%s has registered (%d)", baseptr,
myKey);

  if((qkey = irGetQKey(LSPSIP_PROC)) == IRR_FAIL)  {
    myError("irGetQKey for LSPSIP failed.  EXITING");
    exit(-1);
  }
  irTrace(CHAN, T_LVL, T_AREA, "Qkey for LSPSIP is %d", qkey);

  while(1)  {
    if(irWait() == IRR_FAIL)
```

```
  myError("IrWait failed");

rc = irCheck(&ev);
irTrace(CHAN, T_LVL, T_AREA,
    "irCheck returns %d, event_id = %d", rc, ev.event_id);
if(rc == IRR_FAIL)
  myError("irCheck failed.");

switch(ev.event_id)  {
case IRE_EXEC:
  myQTrace("In IRE_EXEC");
  CHAN = ev.event_mod1;
  if(irInit(CHAN, &cid, IRD_IMMEDIATE, 0) != IRR_OK)  {
    myError("irInit failed.");
    exit(1);
  }

  if(irAnswer(cid) < 0)  {
    myError("irInit failed.");
    exit(1);
  }
  break;

case IRE_CHAN_GRANT:
  myQTrace("Got IRE_CHAN_GRANT");
  break;
case IRE_CHAN_DENY:
  myQTrace("Got IRE_CHAN_DENY");
  break;
```

```
case IRE_DEINIT_DONE:
  myQTrace("Got IRE_DEINIT_DONE");
  exit(0);
case IRE_ANSWER_DONE:
  myQTrace("Got IRE_ANSWER_DONE");
  /*
   * Play a phrase: "Begin Testing"
   */
  irFPlay(cid, 0,
    "/home/sdoliver/lsps/speech/begn_tst.mu8");
  irEnd(cid, 0, 0);
  break;
case IRE_PLAY_DONE:
  myQTrace("Got IRE_PLAY_DONE");
  if(PLAY != 0)  {
    break;
  }
  PLAY++;
  /*
   * setup for ASR and talk to lspsip
   */
  irGetParam(cid, IRP_RECOG_PRETIME, &value);
  irTrace(CHAN, T_LVL, T_AREA,
    "IRP_RECOG_PRETIME = %d\n", value);

  irGetParam(cid, IRP_RECOG_INTERTIME, &value);
  irTrace(CHAN, T_LVL, T_AREA,
    "IRP_RECOG_INTERTIME = %d\n", value);
```

```
    if(irSetParam(cid, IRP_RECOG_GAPTIME, 100) == IRR_FAIL)
      myError("irSetParam(IRP_RECOG_GAPTIME) failed\n");

    if(irSetParam(cid, IRP_RECOG_EFLOOR, 0) == IRR_FAIL)
      myError("irSetParam of IRP_RECOG_TYPE failed\n");

    if(irSetParam(cid, IRP_RECOG_TYPE, IRD_WHOLE_WORD)
      == IRR_FAIL)
      myError("irSetParam of IRP_RECOG_TYPE failed\n");

    if(irSetParamStr(cid, IRP_RECOG_GRAMNAME, "IW_YESNO")
      == IRR_FAIL)
      myError("irSetParam of IRP_RECOG_GRAMNAME failed\n");

    if(irStartRecog(cid, 0) == IRR_FAIL)
      myError("irStartRecog failed\n");

    value = irCheckRecog(cid);
    irTrace(CHAN, T_LVL, T_AREA,
      "irCheckRecog returns = %d\n", value);

    if(irStartRecogTimer(cid) == IRR_FAIL)  {
      myError("irStartRecogTimer failed\n");
    }
    break;
  case IRE_INPUT_DONE:
    myQTrace("Got IRE_INPUT_DONE");
    if((rc = irGetInput(cid, buf, 20)) == IRR_FAIL)  {
      myError("irGetInput failed\n");
```

```
    }
    else  {
      irTrace(CHAN, T_LVL, T_AREA,
      "DATA: %s\n", buf);
    }
    break;
  case IRE_DISCONNECT:
  case IRE_DISCONNECT_DONE:
    myQTrace("Got IRE_DISCONNECT[_DONE]");
    if(irSetEvent(cid, IRE_DEINIT_DONE, IRF_NOTIFY)
      == IRR_FAIL)
      myError("irSetEvent(IRE_DEINIT_DONE) failed.");
    if(irDeinit(cid) < 0)
      myError("irDeinit failed.");
    break;
  case IRE_EXTERNAL:
    myQTrace("Got IRE_EXTERNAL");
    hdptr = (struct mbhdr *) ev.event_text;
    switch(hdptr->mcont)   {
    case 100:
      send_msg = (struct send_msg *) ev.event_text;
      irTrace(CHAN, T_LVL, T_AREA,"best choice: %s\n",
        send_msg->data.tsm.best_choice);
      break;
    case LSPS_ASR:
    case LSPS_RECOGCOMP:
      stat = (struct asrStat *) ev.event_text;
      irTrace(CHAN, T_LVL, T_AREA,
        "ASR returns: %s\n",stat->results[0].text);
```

```
irTrace(CHAN, T_LVL, T_AREA,
  "timeslot: %d\n",stat->timeSlot);
irTrace(CHAN, T_LVL, T_AREA,
  "asrResource: %d\n",stat->asrResource);
speak_return(stat->results[0].text,
  stat->hd.irChan, qkey);

strcpy(recv_script.sr_type, "CD_4");
recv_script.sr_sp_brdidx = stat->asrResource;
recv_script.sr_min = 4;
recv_script.sr_max = 4;
strcpy(recv_script.sr_input,
  stat->results[0].text);
recv_script.sr_recogz = 2;
strcpy(recv_script.sr_expect,
  stat->results[0].text);
recv_msg.mhdr.mcont = 100;
recv_msg.mhdr.mchan = CHAN;
recv_msg.mhdr.irChan = CHAN;
memcpy(&(recv_msg.data), &recv_script,
  sizeof(struct recv_script));
if((recog_qkey = irGetQKey(SP_RECOG_DIP)) ==
  IRR_FAIL)  {
  myError("irGetQKey(recog_dip) failed");
}
rc = irPostEventQ(recog_qkey, &recv_msg,
  sizeof(struct recv_msg));
if(rc == IRR_FAIL)
  irTrace(CHAN, T_LVL, T_AREA,
```

```
            "irPostEventQ failed\n");
          break;
        default:
          irTrace(CHAN, T_LVL, T_AREA,
            "unexpected mcont = %d\n",hdptr->mcont);
          break;
        }
        break;
      default:
        irTrace(CHAN, T_LVL, T_AREA, "Unexpected event %d\n",
          ev.event_id);
        break;

      }

    }

  }

/*************************************************************
* NAME: myError(string)
*
* FUNCTION:
* An error has occured.  Create a trace message using the string
* send and the error number retrieved from the extern int irError.
*************************************************************
*/
void myError(string)
```

```
char * string;
{
 irTrace(CHAN, T_LVL, T_AREA, "%s: %s (%s:%d)",
     string, irErrorStr(irError), irErrorName(irError), irError);
 return;

}
void myQTrace(string)
char * string;
{
 irTrace(CHAN, T_LVL, T_AREA, "%s", string);
 return;
}
void
speak_return(char * text, int chan, ir_key_t qkey)
{
 char string[80];
 int i;

 for(i = 0; i < (int) strlen(text); i++)  {
   /* Setup the files to play */
   sprintf(string, "%s%c%s", "/home/sdoliver/lsps/speech/alph_",
          text[i], ".mu8");
   irFPlay(cid, 0, string);
   irTrace(CHAN, T_LVL, T_AREA, "playing %s\n", string);
 }
 irEnd(cid, 0, 0);
}
```

Fax Examples

Fax Print

```
#include        <sys/types.h>
#include        <stdlib.h>
#include        <libgen.h>
#include <stdio.h>
#include <irapi.h>
#include <iriAPI.h>

#define TIFF_PRINT_FILE "/vs/data/testfax"


/*
* Simple FAX call and answer test program:
*
* Outcall:
*  Upon invocation with arguments:
*   - init a channel
*   - outcall
*   - print FAX
*   - hangup
*   - deinit
*   - exit
*
```

```
*
*/

int Error = 0;

void cleanup(const char *string, channel_id cid)
{
   Error = 1;

   irPError(string);

    void) irDeinit(cid);
}

void startTest(channel_id cid, char *number)
{
    if ( 0
      || irSetEvent(cid, IRE_DEINIT_DONE, IRF_NOTIFY) == IRR_FAIL
      || irCall(cid, 0, number) == IRR_FAIL
    {
           cleanup("startTest failed.", cid);
           return;
    }
    return;
}

main(int argc, char *argv[])
{
    ir_key_t qid;
    ir_event_t ev;
```

```
channel_id cid;
int chan;
char *number;
int ret;

if ((qid=irRegister(basename(argv[0]))) < 0) {
    irPError ("Error on irRegister");
    exit (1);
}

if ( argc > 1 ) {
   if (argc != 3) {
     fprintf(stderr, "USAGE:%s <channel> <number>\n", argv[0]);
     exit(1);
}
   /* process command line arguments (phone #, fax_file) */
   chan = atoi(argv[1]);
   number = argv[2];
   if((ret = irInit(chan, &cid, 10000, 0))==IRR_FAIL){
        irPError("IrInit failed.");
        exit(1);
   }
   if ( ret == IRR_OK ) startTest(cid, number);
}

while (irWCheck(&ev) != IRR_FAIL) {
   cid = ev.cid;
   chan = irCid2Chan(cid);
   fprintf(stderr,"%s\n",irPrintEvent(&ev));
```

```
switch (ev.event_id) {
    case IRE_CHAN_GRANT:
     startTest(cid, number);
     break;

    case IRE_CHAN_DENY:
      exit(1);
      break;

    case IRE_CALL_DONE:
     if ( ev.event_mod1 != IREM_BLIND ) {
       cleanup("Outcall Failed.", cid);
       break;
     }
     if ( irFAXPrint(cid, TIFF_PRINT_FILE) == IRR_FAIL ) {
       cleanup("irFAXPrintText Failed.", cid);
       break;
     }
     if ( irFAXEnd(cid, "123456789", "Test Fax Print", 3456)
       == IRR_FAIL ) {
       cleanup("irFAXEnd failed.", cid);
     }
      break;

    case IRE_FAXPRINT_DONE:
      fprintf(stderr,"ev.event_mod1=%d ev.event_mod2=%d
       ev.event_mod3=%d " "ev.event_mod4=%d",
       ev.event_mod1, ev.event_mod2, ev.event_mod3,
```

```
                ev.event_mod4);
              Error = (ev.event_mod1 == IREM_COMPLETE) ? 0 : 1;
              (void) irDeinit(cid);
              break;

            case IRE_WINK:
            case IRE_DISCONNECT:
          if ( irDeinit(cid) == IRR_FAIL )
            cleanup("Error on irDeinit", cid);
          break;

            case IRE_DEINIT_DONE:
              exit(Error);
              break;
          }
      }
}
```

Fax Record

```
#include <stdio.h>
#include <irapi.h>
#include <iriAPI.h>
#include <libgen.h>

#define RECORD_FILE"/tmp/faxRecord.out"
```

```
#define MAX_BYTES 0  /* max length of fax in bytes*
                     * <= 0 implies unlimited.*/

void cleanup(const char *string, channel_id cid)
{
   int chan = irCid2Chan(cid);
   irPError(string);
   void) irDeinit(cid);
}

main(int argc, char *argv[])
{
   ir_key_t qid;
   ir_event_t ev;
   channel_id cid;
   int chan;
   int max_bytes = MAX_BYTES;

   if ((qid=irRegister(basename(argv[0]))) < 0) {
     irPError ("Error on irRegister");
     exit (1);
   }

   if ( argc > 1 ) {
      max_bytes = atoi(argv[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;
    }
  if (irAnswer(cid) == IRR_FAIL)
    cleanup ("Error on irAnswer", cid);

     break;

    case IRE_ANSWER_DONE:

     if ( irFAXTone(cid, IRF_CNG) == IRR_FAIL )
      cleanup("Error on irFAXTone", cid);

       break;

      case IRE_FAX_TONE:
      if ( irFAXRecord(cid, 0, max_bytes, RECORD_FILE) ==
```

```
                IRR_FAIL )
                cleanup("Error on irFAXRecord", cid);
                break;

              case IRE_FAXRECORD_DONE:
                fprintf(stderr, "ERROR_MODIFIER=%s PAGES=%d BYTES=%d
                  TSI=%s(%d)\n",
                  irErrorName(ev.event_mod2), ev.event_mod3,
                  irGetVCount(cid), ev.event_text, ev.event_text_len);

                if ( irDeinit(cid) == IRR_FAIL )
                  cleanup("Error on irDeinit", cid);

                break;

              case IRE_WINK:
              case IRE_DISCONNECT:
                if ( irLibState(cid) != IRS_FAXRECORDING ) {
                if ( irDeinit(cid) == IRR_FAIL )
                  cleanup("Error on irDeinit", cid);
            } /* else ignore and deinit when fax is done. */
              break;
              }
          }
}
```

B Summary of TAS Script Instructions

Overview

This appendix contains more detailed information about the transaction assembler script (TAS) instructions discussed in Chapter 3, TAS Script Instructions , and Chapter 4, Data Interface Processes , of this book.

The TAS script instructions are listed in alphabetical order. Each script instruction is on a separate page, with the following information provided:

- Instruction name and syntax
- Purpose of the instruction
- What the instruction does
- Examples of the instruction

TAS Script Instruction Syntax

In presenting the syntax for a TAS script instruction, the following conventions are used:

- The script instructions are displayed in **bold** type.
- Associated options are displayed in ***bold italic*** type.
- Examples are shown in `constant-width` type.
- Mandatory arguments or identifiers are displayed within parentheses — ( ).
- Optional arguments or identifiers are displayed within brackets — [ ].
- Lists of options for a single argument are divided by pipe symbols — |, for example, a|b|c|d.

and

Name

The **and** instruction implements an AND operation on the specified arguments.

Synopsis

and(*type.dst,type.src*)

Description

The **and** instruction implements a bitwise AND operation on the arguments. The results are stored in *type.dst.*

Example

The following example clears the bits not set in FLAG in r.3.

```
and(r.3,FLAG)
```

atoi

Name

The **atoi** instruction converts an ASCII string to an integer.

Synopsis

atoi*(type.dst,ctype.src)*

Description

The **atoi** instruction converts a null-terminated character string at the *ctype.src* to an integer value and stores that value at the *type.dst*. If an error occurs, the **atoi** instruction returns a 0 in *type.dst*.

Example

The following example converts a null-terminated character string found at the address labeled ISIZE to a numeric value and puts it in *r.1*.

```
atoi(r.1,ch.ISIZE)
```

background

Name

The **background** instruction starts and/or listens to background audio on the specified channel.

Synopsis

background(*"phrase_name",type.src,[type.style]*)

background(*type.src,type.src,[type.style]*)

Description

Note: A time division multiplexor (TDM) bus and a speech and signal processor (SSP) or LSPS II circuit card must be installed in the system for the background instruction to function properly. Play must be assigned to the appropriate circuit card.

The **background** instruction starts and/or listens to background audio on the specified channel. The first argument is a phrase enclosed in quotation marks (" "). The phrase must match a phrase listed in the talkfile specified by the currently active tfile instruction. The first argument can translate also to the index number of a phrase in the talkfile. In this case, the argument must be expressed according to the conventions of *type.src*. This syntax is similar to the **talk** instruction, but only supports one phrase rather than a phrase list.

If this phrase is not playing already in the system, it is started and its audio output added to the normal voice response prompts on the current channel. Other channels may execute the same background instructions. The audio then is added to those channels while it still is played on the first channel. When the phrase has been played, it starts again at the beginning. The phrase continues to play as long as at least one channel requires it. The background audio stops when all channels requesting it have dropped it. Background speech plays at a volume level that is 33 percent of foreground speech for the SSP circuit card. Background speech on the LSPS II circuit card must be recorded at lower levels because the LSPS II circuit card does not provide gain control.

The LSPS II does not provide gain control. Therefore, background speech to be played on the LSPS II must be recorded at a lower volume.

If the **background** instruction is successful, it returns 0 in *r.0*. If the instruction is not successful, it returns a negative value in *r.0*.

The following are possible reasons the **background** instruction might fail:

- An attempt was made to add more than one background audio to a channel
- Channel reached the limit for listen time slots (maximum of seven per channel)
- No SSP or LSPS II circuit card is available
- All TDM slots are busy

- Reached system limit on number of backgrounds (MAXCHAN)
- System call failure

Example

```
#define ADD 1
#define DROP 0

tfile("/speech/talk/list.cabnt")
background("begin testing",ADD)
background(201,DROP)
```

case

Name

The **case** instruction calls a function if the values are equal.

Synopsis

case(*type.src,type.src[,<subroutine_label>]<goto_label>*)
case*(type.src,type.src[,<subroutine_label> ()]<goto_label>*)
case(*type.src,type.src[,<subroutine_label>(type.src)]<goto_label>*)
case(type.src,type.src[,<subroutine_label>](type.src,type.src)<goto_label>)

Description

The **case** instruction provides a conditional subroutine call that compares two source values. If they are equal, the subroutine is called, and on return, execution continues at the *goto_label* address. If they are unequal, the statement is treated as a no-op instruction and execution continues. If the *subroutine_label* is -1, no subroutine call is made and execution continues at the *goto_label*. If the *goto_label* is -1, execution continues with the next instruction.

As is normal for subroutine calls, calling the specified subroutine saves the values of all registers except *r.0*. Register 3 contains the first optional subroutine argument and register 2 contains the second optional subroutine argument.

Example

Based on the value of int.FOLLOW_UP, one of two telephone numbers is dialed.

```
case(int.FOLLOW_UP,F_ATD_A,CALL(APHONE),w4answ)
case(int.FOLLOW_UP,F_ATD_B,CALL(BPHONE),w4answ)
...
w4answ:
...


CALL: /* Call an attendant. Phone number is at address in r.3 */
   tic('o', int.ATDTIC, *ch.3)
   rts()
```

chantype

Name

This script instruction enables scripts to determine on which type of channel they are running.

Synopsis

chantype ()

Description

The **chantype** instruction returns in *r.0* the following positive integer values from the **/att/include/irDefines.h** header file (Table 22):

Table 22. chantype Return Values

Value	Channel Type
IRD_T1	T1 (E&M) or E1 (CAS) protocol
IRD_PRI	IDSN PRI protocol
IRD_VIRT_CHAN	Virtual channel

A negative value is returned if an error occurs.

Example

```
#include “/att/include/irDefines.h”

/* get channel type */
chantype()
load(int.F_chantype, r.0)
```

```
/* channel type must be T1 or PRI */
jmp(int.F_chantype == IRD_T1, L__chan_OK)
jmp(int.F_chantype == IRD_PRI, L__chan_OK)
```

dbase

Name

The **dbase** instruction sends a message to a data interface process (DIP).

Synopsis

dbase*(type.dip,mcont_field,ctype.dst,mbyte,type.src,nbyte)*

Description

The **dbase** instruction sends a message to a DIP and usually receives data in return. It uses any DIP to interface with the host or local database. All the arguments must be specified for the **dbase** instruction to execute. The arguments are defined by the script writer. See Chapter 4, Data Interface Processes, for more information.

A message is sent to a DIP specified by the first argument in the **dbase** instruction. The *type.dip* argument can be a DIP number (for hardcoded DIPs) or name (for Dynamic DIPs). The *mcont_field* is a DIP-specific code signifying the DIP action. The information returned by the DIP is stored at the destination address specified by *ctype.dst*; its length is specified by *mbyte*. If *mbyte* is negative, the **dbase** call will not wait for a response from the DIP. The information passed to the DIP from the TSM is read beginning from the address specified by *type.src*; its length is specified by *nbyte*. It is important that the DIP and TSM script agree on the structure and contents of the

information passed. If the **dbase** call is successful and the DIP returns a message to the script, *r.0* is set to the *mcont* value of the DIP message.

If *type.src* is a register, *nbyte* is ignored. If *nbyte* is zero, no information is passed to the DIP. If *nbyte* is negative, no message is sent to the DIP, but the **dbase** call may wait (if *mbyte* is not negative) for a message from the DIP. If the DIP is not running, *r.0* is set to -1. If the DIP does not respond within a reasonable time (the default value is 45 seconds), *r.0* is set to -2. To reset the default value for timeout, use the **nwitime** instruction.

Example

In the following example, this instruction uses DIP "Bankdip" to retrieve information. The DIP action is defined by the second argument (LOGON). The retrieved information is stored in user memory beginning at CUSTRECS and has a length of SZCUSTREC bytes. CUSTRECS is found in the ***application-name*.def.h** header file. The argument LOGONS is also defined in the ***application-name*.def.h** header file and marks the starting address of the information passed to the DIP for retrieving the information. The LOGONS field is 5 bytes long.

```
dbase("Bankdip",LOGON,ch.CUSTRECS,SZCUSTREC,ch.LOGONS,5)
```

See Also

getdig
getinput

decr

Name The **decr** instruction decreases a value.

Synopsis **decr(*type.dst,type.src*)**

Description The **decr** instruction decrements the *type.dst* value by the *type.src* value.

Example The following example decreases r.3 by the value defined by NSTKS.

```
decr(r.3,NSTKS)
```

dipname

Name The **dipname** instruction translates a DIP number to a DIP name.

Synopsis **dipname**(***ctype.dst, type.src***)

Description The **dipname** instruction stores the DIP name in *ctype.dst* corresponding to the TSM DIP number specified in *type.src*. *Ctype.dst* should be least BNAMELENG (as defined in **shmemtab.h**) bytes long. The **dipname** instruction stores a null string if the DIP number:

- Is not between 1–34 or 44–75
- Does not have an associated message queue already created

- Maps to a message queue key that is not assigned to a DIP

Note that the contents of the registers is not affected by this instruction.

The **dipname** instruction is used mostly for scripts that catch the DIP interrupt event and need to translate the DIP number of interrupting DIP to a DIP name.

Examples

```
/* Space for DIP name */
#define   DIPNAME       30
dipname(ch.DIPNAME,r.1) /*DIP number in register 1 */
dipname(ch.DIPNAME,0) /* DIP number 0 */

dipname(ch.DIPNAME,int12) /* DIP number in integer location 12
*/
```

See Also **dipnum**

dipnum

Name The **dipnum** instruction translates a DIP name to a DIP number

Synopsis **dipnum**(***type.dst,ctype.src***)

Description

The **dipnum** instruction stores the DIP number in *type.dst* corresponding to the TSM DIP name specified in *ctype.src*. The **dipnum** instruction stores a -1 if the DIP name:

- Is invalid
- Does not have an associated message queue already created
- Maps to a message queue key that is not assigned to a DIP.

Note that the contents of the registers is not affected by this instruction.

Examples

```
/* Space for DIP name */
#define    DIPNAME 30
#define      dipnum 48
dipnum(r.1, ch.DIPNAME)

dipname(int.dipnum, "Dip")
```

See Also

dipname

dipterm

Name

The **dipterm** instruction specifies that a DIP receives a message when the script terminates.

Synopsis

dipterm(***type.dip[,flag]***)

Description

The **dipterm** instruction specifies to TSM which DIP receives a termination message when the script terminates. A DIP number or name may be used for *type.dip*. The **dipterm** instruction may be called repeatedly with different DIP numbers or names. The termination message goes to all DIPs specified.

The optional *flag* may be used to turn off a **dipterm** setting. The valid values for the *flag* are

1 — Set **dipterm** for *dip* (default)

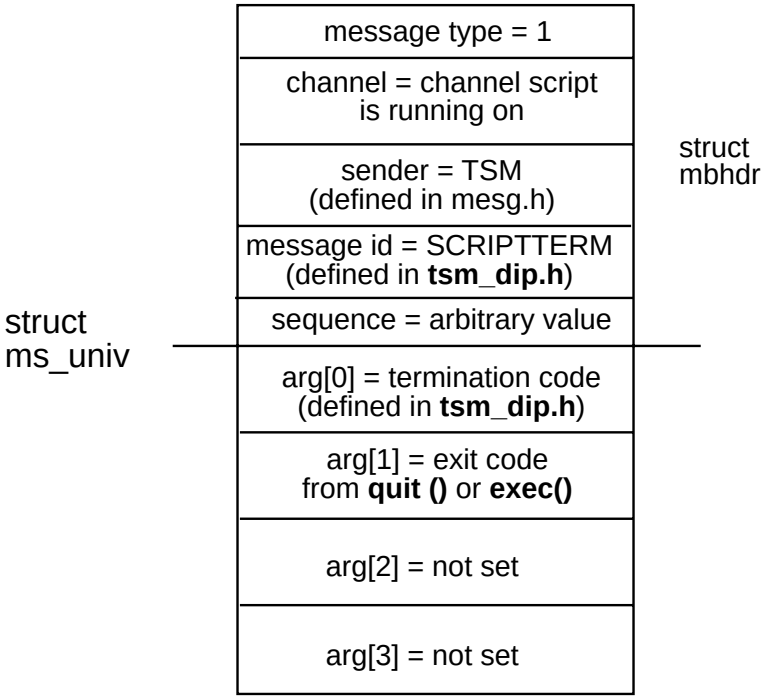
0 — Reset **dipterm** for *dip*

The **dipterm** message is defined as the C-structure struct **ms_univ** (see **mesg.h**). Figure 29 and Figure 30 on page 437 show the fields of the message and their values as set by TSM.

Figure 29. dipterm Synopsis

```
/* message structure for dipterm message */
struct ms_univ {
   struct mbhdr     hd;
   long             arg[4];
   };
```

Figure 30. dipterm Message Structure



arg[0], as shown, displays why the script terminated. As defined in **tsm_dip.h**, there are several causes for a script to terminate.

NORMALTERM	A quit instruction in the script was executed.
DISCONTERM	The call was disconnected.
SCRFAILTERM	An error exists in the script code.
MTCTERM	The MTC process seized the channel on which the script is running.
EXECTERM	The script exec'ed another script.

arg[1] is set to the value specified in the **quit** or **exec** instructions.

Example

The following example causes DIP0 to receive a termination message when the script terminates.

```
dipterm(0)
```

The following example causes the DIP called "bankdip" to receive a termination message when the script terminates.

```
dipterm("bankdip")
```

div

Name

The **div** instruction divides a value.

Synopsis

div(***type.dst,type.src***)

Description

The **div** instruction divides the *type.dst* value by the *type.src* value. The integer quotient is returned in *type.dst*. The **div** instruction returns a value of 0 (zero) in Register 0 if no error occurred. If division by 0 is done and a -1 value is returned in Register 0, the result is set to the largest positive or negative integer, depending on whether *type.dst* was positive or negative originally.

Example

The following example divides r.3 by the value defined by NSTKS.

```
div(r.3,NSTKS)
```

dtitos

Name

The **dtitos** instruction converts the date and time from an internal form to the "tm" structure form.

Synopsis

dtitos(*type.src, type.dst*)

Description

The **dtitos** instruction converts the date and time from the internal UnixWare system representation to "tm" structure form. The *type.src* argument should contain a number representing the UnixWare system internal representation of time (number of seconds since 00:00:00 GMT, January 1, 1970). It is recommended that the integer type be used for this argument. The resulting "tm" structure (the nine-integer structure defined in CTIME(3C) in the *UnixWare Programmer's Reference Manual*) is put in *type.dst* (that is, *type.dst* defines a starting address for the result).

The dtitos instruction returns 0 in script register 0 (*r.0*) if the conversion is successful. *r.0* contains -2 if TSM could not allocate enough space in script memory to store the result.

Example

In the following example, the script plays the system date and time, then says "good-bye" and hangs up. Note that phrase numbers for the days of the week and month of the year in the stdspch.pl play file are offset from "sunday" and "january" by the values obtained in TM_WDAY and TM_MON, respectively.

```
#define TM 8
#define TM_SEC 8/* seconds after the minute (0-59) */
#define TM_MIN12/* minutes after the hour (0-59) */
#define TM_HOUR  16/* hour since midnight (0-23) */
#define TM_MDAY 20/* day of the month (1-31) */

#define TM_MON 24/* months since January (0-11) */
#define TM_YEAR 28/* years since 1900 */
#define TM_WDAY 32/* days since Sunday (0-6) */
#define TM_YDAY 36/* days since January 1st (0-365) */
#define TM_ISDST 40/* flag for daylight savings time */
                    /* (non-zero if alt. timezone in effect) */

#define WKDAYPH      44
#define MONTHPH      48


tfile("stdspch.pl")
dtitos(time.0, ch.TM)
tic('a')
talk("date")
load(int.WKDAYPH, "sunday?")
incr(int.WKDAYPH, int.TM_WDAY)
talk(int.WKDAYPH)
load(int.MONTHPH, "january")
incr(int.MONTHPH, int.TM_MON)
talk(int.MONTHPH)
tnum(int.TM_MDAY)
tnum(19)
```

```
tnum(int.TM_YEAR)
sleep(2)
talk("time")
tnum(int.TM_HOUR)
tnum(int.TM_MIN)
sleep(2)
talk("goodbye")
quit()
```

See Also **dtstoi**

dtstoi

Name The **dtstoi** instruction converts the date and time from the “tm” structure to a UnixWare system internal form.

Synopsis **dtstoi(*type.src, type.dst*)**

Description The **dtstoi** instruction converts the “tm” structure specified by the *type.src* argument and converts it to the internal UnixWare system representation. The result is placed in *type.dst*. An integer type should be used for *type.dst*. This instruction is the complement to the **dtitos** instruction.

The **dtstoi** instruction returns 0 in script register 0 (*r.0*) if the conversion was successful. A value of -1 is returned in *r.0* if the “tm” structure indicated by

type.src contains incorrect values or is at a location outside the script data area.

Example

In the following example, the script fragment gets the current system date and time, truncates the hour to midnight, and converts the result back to UnixWare system time stored at location MIDNGT.

```
#define TM  8
#define TM_SEC 8/* seconds after the minute (0-59) */
#define TM_MIN 12/* minutes after the hour (0-59) */
#define TM_HOUR 16/* hour since midnight (0-23) */
#define TM_MDAY 20/* day of the month (1-31) */
#define TM_MON 24/* months since January (0-11) */
#define TM_YEAR 28/* years since 1900 */
#define TM_WDAY 32/* days since Sunday (0-6) */
#define TM_YDAY 36/* days since January 1st (0-365) */
#define TM_ISDST 40/* flag for daylight savings time */
       /* (non-zero if alt. timezone in effect) */
#define MIDNGT 44
...
dtitos(time.0, ch.TM)
load(int.TM_HOUR, 0)
dtstoi(ch.TM, int.MIDNGT)
```

See Also

dtitos

event

Name

The **event** instruction causes a subroutine call when defined events occur.

Synopsis

event(*event_type*[, *subroutine_label*])

event(*event_type*[, *type.offset*])

Description

The **event** script instruction causes a jump to the *subroutine_label* given when events defined by the *event_type* argument occur. The event types are defined in the header file **/att/msgipc/tsm_dip.h**.

If valid arguments are passed, the **event** instruction returns an integer offset in register 0 (*r.0*). This offset is the value of the previous *subroutine_label* (if any) used for the event. It may be saved and used later as the *type.offset* argument to the **event** instruction to reset the *subroutine_label* back to its previous value. (This is useful for external script functions that need to handle events and want to restore their disposition to whatever the calling script had set before returning.)

If *event_type* is not valid or *type.offset* is larger than the text space of the script, a value of -3 will be returned by the **event** instruction.

A negative value for *type.offset* may be used to set no subroutine label for an event, causing the default action to be taken when the event occurs (see below). If no *subroutine_label* or offset is given, the event instruction returns

in *r.0* the value of the *subroutine_label* currently being used (or -1 if none) without changing the disposition for the event.

The event types are as follows:

- EHANGUP — Hangup event. This event is triggered when dial tone, no loop current, disconnect, or glare conditions are detected on the channel. The register value passed to the event subroutine is EHANGUP for *r.0*. If no event subroutine is set for this event, the script exits as if the quit instruction were used.
- EDIALTONE and ESTUTTERDT— Dial tone event. These are special cases of the EHANGUP event. Normally, EHANGUP is triggered when dial tone or stutter dial tone is detected (and the script is not expecting dial tone). EDIALTONE and ESTUTTERDT are used to treat dial tone detection separately from EHANGUP. If both EHANGUP and EDIALTONE/ESTUTTERDT are set with the **event** instruction to call different interrupt routines, EDIALTONE/ESTUTTERDT must be set following EHANGUP.

  The register value passed to the event subroutine is EDIALTONE for *r.0*. If no event subroutine is set for this event, the script exits as if the **quit** instruction was used.

- ESOFTDISC — Soft disconnect event. This event is triggered by sending a SOFT_DISC message to TSM from a DIP (see **/att/msgipc/tsm_dip.h**). This message is acknowledged with a SOFT_DPASS message before the event subroutine is called. Note that if the channel specified by the SOFT_DISC message is idle, a SOFT_DFAIL message is returned.

  If an event subroutine is set, it receives the following values when the event occurs:

r.0	Event type (ESOFTDISC)
r.1	Value from *arg[1]* of SOFT_DISC message
r.2	Value from *arg[2]* of SOFT_DISC message
r.3	Number of the DIP that sent the SOFT_DISC message

  If no event subroutine is set for this event, the script exits as if the quit instruction were used.

- EDIPINT — DIP interrupt event. This event can be triggered by sending a DIP_INT message from a DIP to TSM (see **/att/msgipc/tsm_dip.h**). The DIP_INT message is not acknowledged.

If an event subroutine is set, it will receive the following values when the event occurs:

r.0	Event type (EDIPINT)
r.1	Value from *arg[1]* of DIP_INT message
r.2	Value from arg[2] of DIP_INT message
r.3	Number of the DIP that sent the DIP_INT message

If no event subroutine is set for EDIPINT, TSM ignores the DIP_INT message and the script continues to run.

- ETTREC — Touch tone received event. This event can be used to allow a **dbase**, **sleep**, **tflush**, **talkresume** or **tic** instruction to be interrupted if a touch tone is received while they are being executed. Note: The **tflush** instruction is only interrupted if its first argument is 1 (“talkoff” is disabled).

If an event subroutine is set, it receives the following values when the event occurs:

r.0	Event type (ETTREC)
r.1	Touch-tone character that caused the interrupt
r.2	Number of touch tones received since last **getinput, getdig,** or **ttclear**
r.3	Instruction interrupted 't' - **tflush** or **talkresume**, 's' - **sleep**, 'd' - **dbase**, 'i' - **tic**

If no event subroutine is set for ETTREC, the instructions are not interrupted by touch tones.

- EANSSUP — Answer supervision event. This event is triggered when answer supervision is detected for an E1, T1, or PRI channel.

  The register value passed to the event subroutine is EANSSUP for *r.0*. If no event subroutine is set for this event, the event is not triggered and the script continues to run.

- ERESOURCE — Resource removed event. This event is triggered when a resource that has been explicitly allocated with the **resource_alloc()** instruction is removed due to a maintenance process.

If an event subroutine is set, it receives the following values when an event occurs:

r.0 Event type (ERESOURCE)
r.1 Removed resource capability value (see **resource_alloc()**)
r.2 Removed resource implementation value (see **resource_alloc()**)

If no event subroutine is set for ERESOURCE, the script will not be notified of removal of explicitly allocated resources.

Note: If an explicitly allocated resource is removed, the system will still attempt to dynamically allocate the resource for the script on an "as needed" basis (as if the **resource_alloc()** instruction were never used).

The DIP number stored in *r.3* for ESOFTDISC and EDIPINT events is the same value used by the **dbase** and **dipterm** instructions. It can be used directly by those instructions in the event subroutines, if desired.

Return from an event subroutine is handled the same for all events. If the event routine causes a wait condition, any previous wait condition will be forgotten. If the event routine sets *r.0* to a negative value before returning (with the **rts** instruction), any previous wait condition will be aborted. The wait causing instruction then returns immediately with *r.0* still set to that negative value. In most cases, this simulates a failure condition for the interrupted instruction. If *r.0* is not negative when the event routine returns, the script continues to wait for the expected condition before it continues. When the event routine returns, all registers except *r.0* are restored to the values they

had before the event. Events of different types may be nested. A new event is ignored if an event of the same type is being handled already. The EDIALTONE event also is ignored while EHANGUP is being handled.

Examples

Example 1

The following example shows when a hangup is detected, the script calls the subroutine hangup which records the time in event data space and exits.

```
#define ATD_TIME 24

MAIN:
   tfile("list.atd_mgmt")
   event(EHANGUP, hangup)
   ...

hangup:
   load(ev.ATD_TIME, time.0) /*Record time attendant becomes
free*/
   ...
   quit()
```

Example 2

The following example shows that when a touch tone is detected during the **tflush(1)** instruction, the script stops the play only if the touch-tone digit is a '#'. Note that any received digits are not removed from the script's touch-tone buffer unless a **getinput**, **getdig**, or **ttclear** instruction is done.

```
MAIN:
    ...
    event(ETTREC, L_tkoff)
    talk("something")
    tflush(1)
    /* tflush will return a -5 in r.0 if talkoff (# only) */
    ...
    event(ETTREC, -1)       /* reset event */

    ...

L_tkoff
    jmp(r.3 !='t', L_notkoff)  /* not tflush */
    jmp(r.1 !='#', L_notkoff)  /* not # digit */
    tstop()/* stop play */
    load(r.0, -5)   /* abort tflush() */
    rts()
L_notkoff:
    load(r.0, 0)  /* continue instruction wait */
    rts()
```

exec

Name The **exec** instruction allows a script to start another script.

Synopsis **exec(*ctype.src*[,*type.data*,*type.nbytes*][,*exitval*])**

Description

The **exec** instruction allows a script to execute another script.

The *ctype.src* argument is the name of the script to be executed. The *type.data* and *type.nbytes* arguments are used to pass a block of data to the new script. The *type.data* argument specifies the location of the data, and *type.nbytes* specifies the size in bytes of that data. If *type.data* is a register or immediate type, *type.nbytes* is ignored and the size of an integer (4 bytes) is assumed. These two optional arguments work like the last two arguments of the **dbase** instruction. The *exitval* argument is an optional exit value that will be used when the "parent" script is terminated before the new "child" script is run. It is used in the same way as the argument to the **quit** script instruction and may be specified without specifying the *type.data* and *type.nbytes* arguments. If no *exitval* is given, -1 is used by default.

The **exec** instruction only returns if the script name specified is invalid or the size of the data being passed exceeds the 2-Kbyte default limit. In these cases, register 0 is set to -1. Otherwise, the script exits and the following actions are performed:

- If the *exitval* used is 0 or negative or if no *exitval* is given, a CALLDATA message is sent to CDH (as is done when the **quit** instruction is called). However, if the *exitval* is greater than 0, the CALLDATA record is not written to CDH. In this case, the start time of the call is preserved for the next script, which may send the record out when it executes another **exec** or a **quit**. CALLDATA events cannot be preserved across an **exec** since each script may define those events differently.

- SCRIPTTERM messages are sent to all DIPs for which the **dipterm** instruction was executed by the script. An array of four long integers is passed as data with the SCRIPTTERM message. The first of these is set to NORMALTERM in the case of **quit** but is set to EXECTERM in the case of **exec** (see **tsm_dip.h**). The second integer in the array is set to whatever value is given to the **quit** instruction or in the *exitval* argument to the **exec** instruction. In each case, it is set to -1 if this value is not provided.

Normally, TSM sets all script registers to zero (0) when a new script starts. When a script is run with **exec**, however, the register values set by the old script are preserved for the new script. If any speech has been queued with the **talk**, **tnum**, **tchar**, or **say** instruction, the **exec** causes this speech to be played before the new script is executed.

The system monitor shows the transition when a new script is executed by displaying the new script name under the “Voice Service” heading for the channel. The number under the “Calls Today” heading is not incremented when a new script is started with **exec** unless the new script executes a **tic(’a’)** instruction.

As was mentioned previously, the optional second and third arguments to **exec** may be used to pass a block of data to the new script. This data is not stored in the user data space of the script because that space is usually freed and the new script’s data space takes its place. This means that the new script cannot access the passed data directly as a script variable. Instead a new access code argument, “X.0”, was introduced to reference this data and

some existing instructions have been modified to support this code. The X.0 code may be used as the second argument to the **strcpy** instruction to copy the **exec** data into the script's data space. When this argument is used, **strcpy** performs a block copy of the **exec** data to the place specified by the first argument to **strcpy**. Enough space should be set aside by the script to accommodate the data. **Strcpy** uses the size that was passed by the **exec** instruction in copying the data. It does not look for a null character at the end of the data, as is done normally.

The **strcmp** and **strlen** instructions also accept X.0 for their arguments. In this case, **strcmp** does a byte-by-byte comparison using the size of the exec data as a limit (instead of looking for a null-character termination) and returns in register 0 a value with the same meaning as **strcmp** has had previously (that is, a value less than, equal to, or greater than zero depending on whether the data indicated by the first argument is lexicographically less than, equal to, or greater than that indicated by the second argument). **Strlen** simply returns in register 0 the size of the exec data as it was passed to the exec instruction.

The **exec** data also may be passed directly to a DIP by using the X.0 code as the fifth argument to the **dbase** instruction. The sixth argument indicating the size is ignored in this case since TSM will use the size originally passed to exec. The exec instruction similarly supports the X.0 code for its *type.data* argument. The *type.nbytes* argument also is ignored in this case.

These instructions are the only ones that have been modified to support the X.0 argument code. The TAS script compiler has been changed to do some

checking of the arguments to the dbase and strcpy instructions to ensure that X.0 will not be allowed for the first argument of strcpy and the third argument to dbase. There has been no effort made to do such checking for any other instruction; use of X.0 elsewhere may have unpredictable results.

Example

The following example quits the script with an exit value of 1 and starts executing the “test.script” script.

```
exec("test.script",1)
```

See Also

dipterm
execu
subprog
quit

execu

Name

The **execu** instruction allows a script to start another script.

Synopsis

execu(*ctype.script[,type.data,type.nbytes][,exitval]*)

Description

The **execu** instruction has the same format and functionality as **exec**. Using **execu** instead of **exec**, however, causes the new script to inherit the data space of the “parent” script intact. Essentially, this feature allows a script to pass all its data to the new script. For this to be useful, however, the new

script must have its data defined in the same way as the parent script (that is, structures, variables, etc. must be defined for the same locations). The data definition of the new script will be used to overlay the actual data of the parent script.

Example

The following example quits the script with an exit value of 1 and starts executing the “test.script” script.

```
execu("test.script",1)
```

See Also

exec
subprog

extend

Name

The **extend** executes customer defined C language functions on behalf of TAS scripts.

Writing a script that uses this instructions also requires knowledge of C language programming and some basic knowledge of the IRAPI.

Synopsis

#include “irapi.h”
extend(type.src)

Description

The **extend** instruction differs from all other TSM instructions in that customer written C language functions are executed. Moreover, there are many contexts from which the customer C language functions may be executed.

The TSM process executes (optional) TAS instruction specific code in each of the following contexts: at TSM process initialization, at script start up, at script exit, at TSM process termination, upon receipt of SIGUSR1, and during execution of an instruction from within script.

During TSM process initialization the TSM program calls:

irExtendInit(0)

The **irExtendInit**(3IRAPI) function executes all functions in the IRAPI dynamic switch table **Iri_ExtendInit_table** via a call to **irVADynSwitchAll**(3IRAPI).

The return codes from the functions in this dynamic switchtable are always ignored and do not affect the behavior of IRAPI or TSM.

When TSM first starts executing a script the TSM program calls:

irExtendStart(0, *cid*, *script* **)**

where

- *cid* is the channel ID
- *script* is a string name of the script

The **irExtendStart**(3IRAPI) function executes all functions in the IRAPI dynamic switch table **Iri_ExtendStart_table** via a call to **irVADynSwitchAll**(3IRAPI).

The return codes from the functions in this dynamic switch table are always ignored and do not affect the behavior of IRAPI or TSM.

When TSM encounters the extend instruction in a TAS script the TSM program calls:

irExtendExecute(*id*, ***cid***, ***script***, ***reg*** **)**

where

- *id* is an integer value corresponding to the argument passed to the extend instruction
- *cid* is the channel ID on which the script is running
- *script* is a string name of the script
- *reg* is an array of IRD_NUMREG longs corresponding to the TSM registers for this script.

The **irExtendExecute**(3IRAPI) function returns the value returned by the customer defined C-language function. If **irExtendExecute** returns **IRR_OK**, then **extend** sets Register 0 to 0. If **irExtendExecute** does not return **IRR_OK**, then **extend** sets Register 0 to -1.

The **irExtendExecute**(3IRAPI) function executes the function in the IRAPI dynamic switch table **Iri_ExtendExecute_table** that corresponds to id via a call to **irVADynSwitch**(3IRAPI)

The customer defined C-language function may get and set any of the IRD_NUMREG registers passed to it. The TSM program will set Register 0 to either 0 or -1, according to the return code of the C language function executed, therefore that register should not be written to.

The new IRAPI parameter **IRP_EXTEND_BUF** is a character buffer of 2048 bytes. The TAS script instructions **setIRAPIparamstr** and **getIRAPIparamstr** may be used to get or set this IRAPI parameter. The customer defined C-language functions executing in the context of the **extend** instruction may use the IRAPI functions **irSetParamStr** and **irGetParamStr** to get or set this IRAPI parameter.

When TSM terminates a script program the TSM program calls:

irExtendExit(0, *cid*, *script* **)**

where

- *cid* is the channel ID
- *script* is a string name of the script

The **irExtendExit**(3IRAPI) function executes all functions in the IRAPI dynamic switch table **Iri_ExtendExit_table** via a call to **irVADynSwitchAll**(3IRAPI).

The return codes from the functions in this dynamic switch table are always ignored and do not affect the behavior of IRAPI or TSM.

When TSM process quits the TSM program calls:

irExtendQuit(0)

The **irExtendQuit**(3IRAPI) function executes all functions in the IRAPI dynamic switch table **Iri_ExtendQuit_table** via a call to **irVADynSwitchAll**(3IRAPI). The return codes from the functions in this dynamic switch table are always ignored and do not affect the behavior of IRAPI or TSM.

When TSM process receives a SIGUSR1 the TSM program calls:

irExtendTrace(0)

The **irExtendTrace**(3IRAPI) function executes all functions in the IRAPI dynamic switch table **Iri_ExtendTrace_table** via a call to **irVADynSwitchAll**(3IRAPI). The return codes from the functions in this dynamic switch table are always ignored and do not affect the behavior of IRAPI or TSM.

There are some limitations on what can be done within the functions called by the **extend** instruction. TSM is a multi-channel application. Because of this C-language functions written for the extend instruction must be able to execute in the context of many channels.

Memory allocation, initialization and access to data structures may need to be done on a per channel basis. After TSM completes execution of an instruction TSM is free to execute another instruction for that script, or it may suspend execution of the script and execute an instruction in another script. Therefore no assumptions should be made that two extend instructions immediately following each other will indeed be executed immediately one after the other.

There are two other constraints on the C-language functions executed by the **extend** instruction. These functions must not generate IRAPI events. TSM will not know what to do with these events, and they will be discarded. These functions must not consume so much time that they cause TSM to fall behind in processing its event queue. Any time consuming operations, such as database operations or network communication, must not be done with this mechanism. Use a DIP for any time consuming processes; do not use the **extend** instruction.

The primary purpose of the **extend** instruction is to allow TSM to execute fast C-language functions that would otherwise be difficult or impossible to write in the TAS script language.

Lucent Technologies cannot guarantee add-on packages implementing **extend** instructions written by other than Lucent Technologies will work correctly, or without damage to your system.

Examples

In the following section several sample scripts calling **extend** instructions are presented in detail. All these examples are on-line in the directory **/vs/examples/IRAPI**.

Example 1

```
/*
* FUNCTION:
* get_rand      - get the value returned by rand()
*               and store it in the passed in integer buffer
*
* INPUT:        integer buffer to store random number
*
* RETURNS:      return code from extend(TSM)
*/
#include "irapi.h"
DEFARG_COUNT(1)
DEFARG(rand,num,both)  /*passed in via r.3 */
L__get_rand:
    extend( IRX_RAND )  /*set r.1 to rand() */
    load( *int.3, r.1 ) /*set script parameter to r.1 */
    rts()               /*return to script */
```

Example 2

```
/*
* FUNCTION:
* get_lbolt     - get the irLBolt(3IRAPI) value and store it
```

```
*                       in the passed in integer buffer
*
* INPUT:        integer buffer to store irLBolt() return value
*
* RETURNS:      return code from extend(TSM)
*/
#include "irapi.h"
DEFARG_COUNT(1)
DEFARG(lbolt,num,both)  /*passed in via r.3 */
L__get_lbolt:
   extend( IRX_LBOLT ) /*set r.1 to irLBolt() */
   load( *int.3, r.1 ) /*set script parameter to r.1 */
   rts()               /*return to script */
```

Example 3

```
/*
* FUNCTION: get_time - get current time()
*
* OUTPUT:
*    time:         time in seconds since midnight 1/1/70 UTC
*/
#include "irapi.h"
DEFARG_COUNT(1)
DEFARG(time,num,both)   /* passed in via r.3 */
L__get_time:
   extend  ( IRX_TIME )
```

```
    load    ( *int.3, r.1 )
    rts()
```

Example 4

```
/*
* FUNCTION: get_ctime - get the ctime() equivalent of
* the time, e.g., "Thu Sep 25 12:25:48 1997"
*
* INPUT:
*    destination: field that result is placed in
*    time:        time in seconds since midnight 1/1/70 UTC
*/
#include "irapi.h"
DEFARG_COUNT(2)
DEFARG(destination,char,out)    /* passed in via r.3 */
DEFARG(time,num,in)             /* passed in via r.2 */
L__get_ctime:
    load    ( r.1, r.2 )
    extend  ( IRX_CTIME )
    /* get string up to but not including the \n */
    getIRAPIparamstr( IRP_EXTEND_BUF, *ch.3, 24 )
    incr    ( r.3, 24 )
    load    ( r.3, 0 )      /* null terminate the string */
    rts()
```

Install Sample Scripts

Use the following procedure to install the example scripts above:

1. At the UNIX system prompt, enter **stop_vs**
2. Enter **cd /vs/examples/IRAPI**
3. Enter **make -f example.mk libirUTIL.so**
4. Enter **make -f example.mk get_lbolt get_time get_rand get_ctime**
5. Enter **cp libirUTIL.so /usr/lib/libirUTIL.so**
6. Enter **editSPIlibs /usr/lib/libirUTIL.so**
7. Enter **start_vs**

The C-language functions implementing these **extend** instructions are on-line in **/vs/examples/IRAPI/util_fcns.c**

For complete information on how to write C-language functions to implement an **extend** instruction see the following manual pages:

irExtend(3IRAPI)
IrEXTEND(4IRAPI)
IrDEFINES(4IRAPI)
IrPARAMETERS(4IRAPI)
irDynSwitch(3IRAPI)
irRegister(3IRAPI)
irSPIRegister(3IRAPI)

irSPI.libs(4SPI)
editSPIlibs(1SPI)
irName(3IRAPI)
irDefine(3IRAPI)
irErrorStr(3IRAPI)
dlopen(3)
dlsym(3)

fsay

Name

This script instruction plays ASCII text stored in a file.

Synopsis

fsay(ctype.src)

Description

The fsay instruction is similar to the say instruction, however it instructs the system to speak ASCII text stored in a file instead of a buffer.

The *ctype.src* argument specifies the fully qualified or relative pathname of a file of ASCII text to be spoken. The script may pass the pathname as a literally quoted string, or the contents of a null-terminated field. The maximum length of the pathname is 128 characters. There is no limit to the size of the file itself.

The **tflush** instruction may be used to flush the TTS speech and cause the text to be spoken. The first two arguments to **tflush** (the must_hear_flag and

the wait_indicator) have the same effect for text-to-speech as for coded speech. (The third argument to **tflush**, (the *remember_flag*, is not used for TTS). That is, the first argument may be used to disable "talkoff" and the second may be used to play speech and to continue the script without waiting for the play to complete. Normally, TSM waits for a TTS play to complete before going on to the next instruction.

The fsay instruction returns one of the following values in script register 0 (r.0) (Table 23 on page 468).

Table 23. Return values for the fsay instructions

Return Value	Return Explanation
0	The fsay instruction completed successfully
-1	The fsay instruction failed to queue the request
-2	One of the following as occurred: • The length of the full pathname passed the fsay instruction was too large • A relative pathname was passed in and the setfalk instruction was not previously called to set the prepend directory • A relative pathname was passed in and the length of the prepend directory plus the length of the relative pathname was too large

See Also

ftalk
setftalk

ftalk

Name

This script instruction plays coded speech stored in a file.

Synopsis

ftalk(ctype.src[, *type.style*])

Description

The ftalk instruction is similar to the talk instruction, however it instructs the system to play coded speech stored in a file instead of a buffer.

The *ctype.src* argument specifies the fully qualified or relative pathname of a file of coded speech to be spoken. The script may pass the pathname as a literally quoted string, or the contents of a null-terminated field. The maximum length of the pathname is 128 characters. There is no limit to the size of the file itself.

When **ftalk** (as well as **talk, tchars,** and **tnum**) instructions are executed, the system queues phrases in a buffer, but the phrases are not immediately played. Phrases are played under either one of the following two conditions:

1. The script executes a speech-flushing instruction
2. The script executes a **say** or **fsay** instruction

The ftalk instruction returns one of the following values in script register 0 (r.0) (Table 24).

Table 24. Return values for the ftalk instructions

Return Value	Return Explanation
0	The ftalk instruction completed successfully
-1	The ftalk instruction failed to queue the request
-2	One of the following as occurred: • The length of the full pathname passed the ftalk instruction was too large • A relative pathname was passed in and the setfalk instruction was not previously called to set the prepend directory • A relative pathname was passed in and the length of the prepend directory plus the length of the relative pathname was too large

See Also

fsay
setftalk

getinput

Note: The **getinput** instruction replaces the **getdig** instruction. Continued use of the **getdig** instruction is discouraged.This instruction can only be used with the SSP circuit card. It cannot be used for the LSPS II circuit card.

Name

The **getinput** instruction receives touch-tone, dial-pulse, or spoken input from a caller.

Synopsis

getinput(*ctype.dst, type.number*[, *int.recognizer*[, *int.resource*]])

Description

The **getinput** instruction manages the process of starting speech recognizers, flushing speech, and collecting input. The behavior of **getinput** is summarized as follows:

1. Start all recognizers queued for recognition while prompting (see **recog_start**).
2. Flush (start output of) all queued speech or text (see **say**, **talk**, etc.).
3. When speech/text output is complete, start all recognizers queued for start after prompting.
4. If at any point a recognizer successfully reports input, all remaining play and recognition activities are terminated and **getinput** returns with the number of characters placed on the input stream.

The argument *ctype.dst* is a character buffer where input data is to be copied. The argument *type.number* indicates the maximum number of input characters to copy to *ctype.dst.* The optional argument *int.recognizer* indicates the address of the integer value where the recognition type used to collect input is stored. Possible values include 0 for TT input or some positive integer indicating a recognizer such as IRD_WHOLE_WORD (see **recog_ start**). The optional argument *int.resource* indicates the address of an integer where the resource used to perform recognition is stored.

When **getinput** is used with multiple simultaneous recognizers, all other recognizers are terminated upon receipt of input from any source and only the results from the first recognizer are reported back to the application. The **getinput** instruction assumes that DTMF (touch-tone) recognition is always on. DTMF recognition can be inhibited with the use of **ttmask** and **ttintr**.

The **getinput** instruction has a 10-second default initial digit wait time for input. If the caller does not enter a digit within the allotted time period, **getinput** returns the number of digits received before the timeout occurred. Use the **tttime()** instruction to specify desired wait times.

The **getinput** instruction is a wait-causing instruction. Therefore, it automatically forces out any pending or unfinished announcements from this channel.

Return Values

The return value from **getinput** is placed in *r.0* as follows:

- *r.0* > 0 — Indicates that **getinput** successfully received *r.0* digits and copied the input to *ctype.dest*.
- 0 — Indicates an initial timeout occurred for all recognizers.
- -1 — Indicates that all resources are busy. This may include recognition resources or play resources if such operations were previously queued with the **recog_start** or **talk** instructions.
- -2 — Indicates that no resources exists to perform the input operation. This may include recognition resources or play resources if such operations were previously queued with the **recog_start** or **talk** instructions.
- -3 — Indicates a system error has occurred.
- -4 — Failed to access a memory location specified via an argument.

Example

In the following example, the script waits for the caller to enter up to 10 digits, then stores them in ch.ANS. The type of recognition used is stored in int.recognizer.

getinput(ch.ANS, 10, int.recognizer)

See Also

getdig
tttime
recog_start
recog_cntl

getIRAPIparam, getIRAPIparamstr

Name

setIRAPIparam, setIRAPIparamstr, getIRAPIparam, getIRAPIparamstr - set/get the value of a IRAPI library channel-based parameter

Synopsis

#include <*irapi.h*>
getIRAPIparam (*identifier,type.dest*)
getIRAPIparamstr (*identifier, ctype.dest, type.count*)
setIRAPIparam (*identifier, type.value*)
setIRAPIparamstr (*identifier, ctype.string, type.count*)

Description

The **setIRAPIparam** and **setIRAPIparamstr** functions assign values to an IRAPI library parameter, while **getIRAPIparam** and **getIRAPIparamstr** get the current value of a parameter. See IrPARAMETERS(4IRAPI) for a list of valid parameters and their default and legal values.

getIRAPIparamstr 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, **getIRAPIparamstr** ignores any null characters in the parameter data. It also makes no attempt to null terminate the *string*. **setIRAPIparamstr** also ignores null characters and copies the number of bytes for the parameter [specified in IrPARAMETERS(4IRAPI) or through *count*] into the call profile. **setIRAPIparamstr** copies only *count* bytes of data beginning at the address specified by *value*, therefore, *value* need only point to an area of size *count*.

Parameters are preserved across **exec** boundaries.

Return Values

getIRAPIparam, **getIRAPIparamstr**, **setIRAPIparam**, and **setIRAPIparamstr** return 0 in register 0 (*r.0*) if successful and return -1 in register 0 (*r.0*) if an error occurs.

See Also

IRAPI manual pages: **irParams**(3IRAPI), **IrPARAMETERS**(4IRAPI), **irGlobalParam**(3IRAPI)

goto

Name

The **goto** instruction unconditionally branches to a label.

Synopsis

goto(*<label>*)

Description

The **goto** instruction is an unconditional jump to the instruction indicated by the label.

Example

In the following example, the **goto** instruction implements if-then logic to avoid the fall-through condition. As shown in the first block of code, the instruction jumps to no_value if true, but must avoid that block of code if false. A **goto** instruction is also used as a direct path out of a block of code.

```
jmp(r.1 <= 0, no_value)
talk("the value is positive")
goto(next_block)

no_value:
   talk("the value is not positive")
   ..........   ...................   ..............

next_block:
   ........   ..................
```

See Also

jmp
case

hbridge

Name

The **hbridge** instruction directs the current channel to bridge partially to another channel.

Synopsis

hbridge(*type.src,type.src)*

Description

The **hbridge** instruction directs the current channel to bridge partially to another channel. The result is that the audio coming in on the specified channel is heard or dropped by the calling party (current channel). The specified channel does not hear the calling party. The current channel does not hear voice responses or other background audio on the specified channel.

The first *type*.*src* argument is a valid channel number. The second *type*.*src* argument is either 1 to add the specified channel or 0 (zero) to drop the channel. Values for the channel numbers and the add/drop flag follow the conventions for all *type*.*src* arguments.

If the **hbridge** instruction is not successful, a negative value is returned to register 0. The following are conditions under which the **hbridge** instruction may fail:

- Hbridge attempt to the current channel
- Channel reached limit for listen tie slots (7 maximum per channel)
- System call failure

Example

```
#define ADD 1
#define DROP 0
#define OTHCHAN17

hbridge (OTHCHAN,ADD)
hbridge(OTHCHAN,DROP)
```

hundsec

Name

The **hundsec** instruction gets the system time in hundredths of a second.

Synopsis

hundsec(*type.dst*)

Description

The **hundsec** instruction loads the integer *type.dst* with the system time in hundredths of a second.

Note: Do not use the **hundsec** instruction in a loop to insert delays in script execution. Use the **sleep** or **nap** instructions instead.

Example

In the following example, HSEC2 contains the duration of the **dbase** call in hundredths of a second.

```
#define HSEC1 10
#define HSEC2 14

...

hundsec(int.HSEC1)
dbase(DIP, GET_DATA, ch.SENDBUF, 20, ch.RCVBUF, 100)
hundsec(int.HSEC2)
decr(int.HSEC2, int.HSEC1)
```

ibrl

Name

The **ibrl** instruction increments a counter and branches to a label if one is less than the other.

Synopsis

ibrl(*type.dst,type.src,<label>*)

Description

The **ibrl** instruction is intended for loop control by testing for equality of two variables. It determines whether to make another pass through a loop or to execute the next sequential instruction. The destination value is incremented by one, and then compared to the source value. If *type.dst* is less than *type.src*, execution jumps to the labeled instruction.

Example

In the following example, after doing some other tasks, *r.3* is increased by 1 and compared with *r.1*. If *r.3* is less than *r.1*, the loop is repeated at the label SW1_CTRL; otherwise, the next instruction is executed which takes the program to end_loop.

```
SW1_CTRL:

     ..... ......... ..................
    ibrl(r.3,r.1,SW1_CTRL)

end_loop:
```

incr

Name

The **incr** instruction increases a value.

Synopsis

incr(*type.dst,type.src)*

Description

The **incr** instruction increments the *type.dst* value by the *type.src* value.

Example

The following example increases the event counter 2 by 1.

```
incr(ev.2,1)
```

itoa

Name

The **itoa** instruction converts an integer to an ASCII string.

Synopsis

itoa(*ctype*.*dst*,*type*.*src***)**

Description

The **itoa** instruction converts a numeric *type.src* value to a null-terminated character string stored starting at *ctype.dst*.

Example

In the following example, a numeric value in r.2 is written at the address labeled ISIZE as a null-terminated character string.

```
itoa(ch.ISIZE,r.2)
```

jmp

Name

The **jmp** instruction jumps to a label if the condition true.

Synopsis

jmp(*type.src rel_op type.src,<label>*)

Description

The **jmp** instruction is a conditional jump to the labeled instruction. The values of the two source operands are compared as specified by the relational operator.

Example

The following example directs the script to go to the attendant subroutine if ch.0 contains * and to go to the BYE subroutine if ch.0 contains #.

```
jmp(*ch.0=='*',attendant)
jmp(*ch.0=='#',BYE)
```

See Also

goto

label

Name

This is a subroutine call.

Synopsis

<label>([type.src] [,type.src])

Description

The **label( )** subroutine call is used to call a subroutine found at the address indicated by the label. A return address and the values in all registers except *r.0* are saved on a subroutine stack in the calling subroutine. The optional first and second arguments are stored in r.3 and r.2, respectively.

Example

In the following example, the integer variables FIRST and SECOND are set equal to 1 and 2, respectively. The subroutine ADDEM is called with two arguments.

Within ADDEM, the variable SUM is set to zero. Then the value of SUM is incremented by r.3 (which has been assigned the value of FIRST from the

calling routine) and incremented by r.2 (which has been assigned the value of SECOND from the calling routine.

The subroutine return (rts) returns control to the tnum instruction following the ADDEM subroutine call.

```
load(int.FIRST,1)
load(int.SECOND,2)
ADDEM(int.FIRST,SECOND)
tnum(int.SUM)

    ADDEM( )
        load(int.SUM,0)
        incr(int.SUM,r.3)
        incr(int.SUM,r.2)
        rts ( )
```

See Also **case**

listenall

Name The **listenall** instruction listens to all audio input on a specified channel.

Synopsis **listenall**(***type.src, type.src***)

Description

The **listenall** instruction listens to all audio input on a specified channel. Audio input includes normal voice responses to the network. The specified channel does not hear any audio from the current channel. This allows administrators to monitor the channel.

The script with the call to **listenall** must be kept running until the caller is finished monitoring the audio input on the other channel. One way to accomplish this would be to add a call to sleep directly after listenall command.

For example:

```
listenall (45, ADD)
sleep (45)
```

These commands keep the monitor script running for 45 seconds after the script starts. You must determine how long the other channel will be monitored and use the appropriate sleep value.

The first *type.src* argument is a valid channel number. The second *type.src* argument is either 1 to add the channel or 0 (zero) to drop it. These arguments must follow the conventions for *type.src* arguments discussed in Chapter 3, TAS Script Instructions.

If the **listenall** instruction is successful, a positive value is returned to Register 0. If the **listenall** instruction is not successful, a negative value is returned to Register 0.

The following are reasons the **listenall** instruction might fail:

- Attempt to monitor current channel
- Attempt to monitor more than one channel
- Channel reached limit for listen time slots (maximum of 7 per channel)
- System call failure

Note: If the **listenall** instruction hears a dialtone, it will hang up.

Example

```
#define ADD 1
#define DROP0
#define OTHCHAN17

listenall(OTHCHAN,ADD)
listenall(OTHCHAN,DROP)
```

load

Name

This script instruction moves data.

Synopsis

load(type.dst,type.src)

Description

The **load** instruction converts the source value to the data type of the destination and stores it at the destination.

Example In the following example, the 4-byte value defined by the name NSTKS is put in *r.1*. The value 3 is written to *r.2*.

```
load(r.1,NSTKS)
load(r.2,3)
```

mul

Name This script instruction multiples a value.

Synopsis **mul(*type.dst,type.src)***

Description The **mul** instruction multiples the *type.dst* value by the *type.src* value and stores the result in the destination.

Example The following example multiplies the event counter 2 by 4.

```
mul(ev.2,4)
```

nap

Name This script instruction causes the script to sleep.

Synopsis **nap*(type.src)***

Description

The **nap** instruction makes the script do nothing for the specified number of centiseconds (hundredths of a second). See the **event** instruction for TSM events that may interrupt the **nap** instruction before the specified time has passed. Unlike the **sleep** instruction, the **nap** instruction does not flush queued speech before the nap is done.

Example

In the following example, the script dials out on a channel, then waits 15 centiseconds for completion of the dial before continuing.

```
tic('d',int.PHONENBR)
nap(15)
```

See Also

event
sleep

no_rts

Name

This script instruction pops the subroutine stack.

Synopsis

no_rts()

Description

This script instruction is the mechanism for popping the subroutine stack without returning from a subroutine call. The values for all registers are unaffected by **no_rts()**.

Example

In the following example, TSM pops the stack but continues with the next instruction instead of returning from the subroutine. The register tracing shows that register values remain unchanged with **no_rts()**, but they are restored, except for r.0, with rts().

```
   trace(1, "begin")

   load(r.0,  100)
   load(r.15, 115)

   trace(1000,  r.0)
   trace(1015,  r.15)

   SUBR()

   trace(2, "returned from SUBR")

   trace(6000,  r.0)
   trace(6015,  r.15)

   rts()   /* should fail, stack is empty */

   quit()

NORETURN:

   trace(3, "no return from SUBR")
   rts()   /* should fail, stack is empty */
   quit()

SUBR:

   trace(2000,  r.0)
   trace(2015,  r.15)
```

```
load(r.0,  200)
load(r.15, 215)

trace(4000,  r.0)
trace(4015,  r.15)

no_rts()        /* replace with no_rts() to see different
result */

trace(5000,  r.0)
trace(5015,  r.15)

goto(NORETURN)
```

not

Name This script instruction implements a NOT operation on the argument.

Synopsis **not(*type.dst*)**

Description The **not** instruction performs a 1's complement operation on the argument.

Example In the following example, *r.3* is changed to its 1's complement.

```
not(r.3)
```

In the following example, the bits set in FLAG are cleared in *r.3*.

```
load(r.2,FLAG)
not(r.2)
and(r.3,r.2)
```

nwitime

Name

This script instruction specifies the amount of time to wait for the next wait-causing instruction.

Synopsis

nwitime(*type.src*)

Description

The **nwitime** (next wait instruction time) instruction sets the maximum amount of time the script will wait for the completion of the next wait-causing instruction. The argument specifies the number of seconds to wait. Instructions that are affected by **nwitime** are: **background**, **dbase**, **phreserve**, **sr_talkoff**, **tic**, and **tstop**.

Example

In the following example, the **nwitime** instruction specifies the maximum number of seconds the script should wait for host confirmation before continuing.

```
ACCT_BAL:
 nwitime(20)
 --
 --
 (query host - dbase())
 --
 --
 talk("Your account balance is")
 tnum(int.FIVE,'f')
 rts()
```

See Also

dbase
phreserve
tic

or

Name

This script instruction implements an OR operation on the arguments.

Synopsis

or(***type.dst,type.src***)

Description

The **or** instruction implements a bitwise OR operation on the arguments.

Example In the following example, bits set in r.3 or FLAG are set in r.3.

```
or(r.3,FLAG)
```

phremove

Name This script instruction removes a phrase from a talkfile.

Synopsis **phremove(*type.phrase,type.talk*)**

Description The **phremove** instruction removes the phrase specified by the *type.phrase* argument from the talkfile specified by the **type.talk** argument. The valid values for *type.phrase* are 1—65,535. The valid values for *type.talk* are 1—16,383. *Type.phrase* must be a valid phrase id. *Type.talk* may have the value -1. If *type.talk* is -1, then the talkfile id used will be the current talkfile. Do not use a character data type as an argument.

If the **phremove** instruction is successful, it returns the phrase id of the phrase removed in register 0. If the instruction is not successful, it returns a negative value in register 0.

Example In the following example, phrase 205 is removed from talkfile 19.

```
load(sh.TALKID,19)
phremove(205,int.TALKID)
```

In the following example, phrase 117 is removed from talkfile 10.

```
load(ch.TALKF,10)
load(int.PHR,117)
phremove(int.PHR,ch.TALKF)
```

phreserve

Name

This script instruction creates space for a phrase in a talkfile.

Synopsis

phreserve(***type.phrase,type.talk,type.time,type.style***)

Description

The **phreserve** instruction creates an area in a talkfile that is used to store a phrase and specifies the coding style and rate to be used. This phrase is later encoded by the **vc** instruction. The arguments for the **phreserve** instruction are:

- *type.phrase* — Specifies the phrase ID of the phrase to be created (valid range is 1–65,535)
- *type.talk* — Specifies the talkfile ID of the talkfile where the phrase is stored (valid range is 1–255)
- *type.time* — Specifies the amount of space, or time (in seconds), to be reserved for the phrase in the talkfile.

- *type.style* — Specifies the coding style and rate to be used. The coding styles and rates are defined in the header file *codestyle.h*. If the coding style and rate are invalid, the instruction fails. Do not use character data types for any of these arguments.

If *type.phrase* is -1, the system assigns a phrase ID and returns this id in register 1. The phrase id can be used to reference the phrase (for example, in a **talk** instruction) once it has been coded and stored in the talkfile by the **vc** instruction. If *type.talk* is -1, the system selects a default value (255) for the talkfile and returns the id of the selected talkfile in register 0.

Note: If there are two **phreserve** instructions, there must be a **vc** instruction between them or the second **phreserve** instruction will fail.

When both *type.talk* and *type.phrase* are -1, both a phrase ID and talkfile ID are chosen by the system and returned in registers 1 and 0 respectively. Since registers 0 and 1 can be used implicitly to store talkfile and/or phrase IDs, the script writer must take care to save the contents of these registers before the phreserve is executed.

If *type.phrase* matches the phrase ID in the specified talkfile, the existing phrase will be replaced by the new phrase. The value 0 or -1 for the *type.time* argument can be used to indicate that the phreserve instruction should not allocate any space. If enough space is available to store the phrase when coding ends, the phrase will be stored. If there is not enough space, an error message will be issued from the **vc** instruction.

If you add phrase 0 to any talkfile, the phrase is added as phrase 65535 the first time. If the command is executed again, the phrase is added to the talkfile as phrase 65534, then 65533, etc.

If the instruction is successfully completed, the return values are:

- Register 0 = talkfile ID
- Register 1 = phrase ID

If the instruction is not successfully completed, the return value in register 0 is negative.

Note: If the script terminates before **vc** instruction is used, the space allocated to the phrase will be freed and the phrase number will be reused the next time a **phreserve** instruction is executed. A new phrase is not stored in the speech file system until a successful **vc** is performed.

Example

In the following example, an area in talkfile 15 is created to store the phrase. The ID for the phrase is returned in register 1 and then loaded into location int.PHRASE. Since *type.time* is -1, the script writer relies on the system having enough space to store the phrase. The coding style for the phrase is ADPCM 32 Kbs.

```
load(int.TALKID,15)
phreserve(-1,int.TALKID,-1,ADPCM32)
load(int.PHRASE,r.1)
```

In the following example, 10 seconds (using ADPCM 32 Kbs coding) of storage are allocated in talkfile 23 for phrase 8.

```
load(ch.20,8)
phreserve(ch.20,23,10,ADPCM32)
```

quit

Name

This script instruction terminates script instructions.

Synopsis

quit([***value***])

Description

The **quit** instruction specifies the intentional termination of a script. A **dipterm** instruction may be defined before using **quit**, but it is not necessary for quit to execute. If **dipterm** is defined, an optional argument can be used. This optional argument is an integer defined by the script writer. It is sent to the DIP specified in dipterm and is usually used to notify the DIP why the script has quit.

Example

In the following example, TSM is instructed to send a termination message to DIP 0 when the script terminates. The script then executes the **quit** instruction, ending the script.

```
dipterm(0)
quit( )
```

See Also

dipterm
exec

recog_cntl

Name

This script instruction disables or enables a recognizer.

Synopsis

recog_cntl(*type.recognizer, type.command*)

Description

recog_cntl allows an application to control recognizer behavior. The argument *type.recognizer* indicates the recognizer to operate on. The argument *type.command* indicates the operation to perform on the recognizer. The following are valid commands:

0 Disables the recognizer. When a recognizer is disabled, it is not queued for starting when **recog_start** is called, that is, the call to **recog_start** is ignored (will fail) until the recognizer is enabled. This can be useful to disable DPR to save resources when you know your caller is using touch tones.

1 Enables the recognizer. By default, all recognizers are enabled so execution of **recog_cntl** is not necessarily required. When a recognizer is enabled, it is queued for starting when **recog_start** is called.

Return Values

The disposition of the call to **recog_cntl** is indicated through register *r.0* as follows:

0 success

-1 (only in the case where command is zero) indicates that the maximum number of recognizers have been disabled (see IRD_MAX_RECOG in irDefines.h).

-2 command is not implemented.

See Also

recog_start
recog_stop
getinput

recog_init

Name

This script instruction initializes training parameters for a recognizer.

Synopsis

#include <*irDefines.h*>
recog_init(type.recognizer)

Description

The instruction **recog_init** initializes training parameters for a recognizer. The argument *type.recognizer* is the recognizer to be initialized. Valid recognizers include IRD_DIALPULSE, IRD_WHOLE_WORD, and IRD_FLEX_WORD as defined in irDefines.h.

Presently, recognizer initialization is only required for the IRD_DIALPULSE recognizer and, furthermore, only when line conditions change, such as after a call transfer. Otherwise the recognizer training parameters are automatically initialized when the call is started.

Default dial pulse training parameters are used until a dial pulse input of five of higher is received. Then, training parameters are created, saved, and used for the duration of this call.

Return Values

TSM register *r.0* contains 0 on success and a -1 on failure to initialize the specified recognizer.

See Also

recog_start
getinput

recog_start

Name

This script instruction queues a recognizer for starting.

Synopsis

For SSP circuit cards:
#include <*irDefines.h*>
recog_start(type.recognizer, type.recog_type, type.while_prompting)

For LSPS II circuit cards:
#include <irDefines.h>
recog_start(type.recognizer, ctype.gramname, type.while_prompting)

Description

The instruction recog_start queues a recognizer for staring at the next invocation of getinput. The argument type.recognizer specifies the recognizer to queue. Valid values are IRD_FLEX_WORD, IRD_LSPS_FLEX_WORD, IRD_WHOLE_WORD, IRD_LSPS_WHOLE_WORD, and IRD_DIALPULSE as defined in irDefines.h.

The argument for SSP circuit cards, *type.recog_type* specifies the type of recognition to use with this recognizer. For example, a grammar taken from **/att/include/sr_grammar.h**. The argument for the LSPS II circuit cards, *ctype.gramname*, corresponds to the IRP_RECOG_GRAMNAME. Chapter 13 of the LSPS Software Development Kit resident on the system at

/usr/lsps/doc/pdf/sdk.pdf contains a list of existing grammar names. See Grammars on page 304 for information on creating custom grammars.

The argument *type.while_prompting* if true (1), indicates to queue the recognizer to be started before the prompt begins. If false (0), indicates to queue the recognizer to be started after the prompt completes. Note that the echo canceler must have been started earlier via **sr_talkoff** to recognize while prompting. If the echo canceler is not running, the recognizer is not started until after the prompt completes regardless of the value of *type.while_prompting*.

Return Values

The return value from **recog_start** is placed in *r.0* as follows:

0 Recognizer successfully queued for starting.

.-1 Maximum number of recognizers already queued.

-2 Recognizer has been disabled via a prior call to **recog_cntl**.

Examples

SSP circuit card example: The following example illustrates the construction of a prompt that requests the caller to enter a 4 digit number. Caller input is accepted as touch-tone, dial-pulse, or speech. Note that it is not necessary to queue the touch-tone recognizer since it is implicitly always on.

```
talk (“enter your 4 digit PIN”)
recog_start(IRD_DIALPULSE, DP_ndig1_10, 1)
recog_start(IRD_WHOLE_WORD, US_4dig, 1)
getinput(ch.ATTNUM, 4, int.RECOGNIZER, int.MODE)
```

LSPS II circuit card example: The following example illustrates the construction of a prompt that requests the caller to enter a 4 digit number. Caller input is accepted as touch-tone, dial-pulse, or speech. Note that it is not necessary to queue the touch-tone recognizer since it is implicitly always on.

```
talk (10)
recog_start(IRD_LSPS_WHOLE_WORD, “NameList”, 1)
getinput(ch.ATTNUM, 4, int.RECOGNIZER, int.MODE)
```

See Also

recog_cntl
recog_stop
recog_init
getinput

recog_stop

Name

This script instruction unqueues a recognizer.

Synopsis

#include <*irDefines.h*>
recog_stop(*type.recognizer*)

Description

The instruction **recog_stop** unqueues a recognizer previously queued via **recog_start**. The recognizer will not be started upon a subsequent call to **getinput**. The argument *type.recognizer* specifies the recognizer to unqueue.

Return Values

The return value from **recog_stop** is placed in *r.0* as follows:

0 Recognizer successfully unqueued.

-1 Recognizer not previously queued.

See Also

recog_start

resource_alloc

Name

This script instruction allocates or frees licensed system resources.

Synopsis

#include <*irDefines.h*>
resource_alloc(type.flag, type.capability, type.implementation [, *type.returnMode*])

Description

Note: The instruction **resource_alloc** replaces **sp_alloc**. Continued use of **sp_alloc** is discouraged.

The instruction **resource_alloc** explicitly allocates capability *type.capability* of implementation *type.implementation* to the channel upon which the script is executed. This allows applications to proceed forth without concern for resource availability at the expense of inefficient resource utilization (applications may hold resources that they are not using).

The argument *type.flag* if true, indicates to allocate resources, otherwise free resources. The argument *type.capability* indicates the capability to allocate as defined in irResource.h, for example IRC_RECOG, IRC_TTS, etc. The argument *type.implementation* indicates the implementation, as defined in irDefines.h, corresponding to the capability argument. For example, an implementation of IRC_RECOG is IRD_WHOLE_WORD. The argument *type.returnMode* indicates the optional return mode on resource allocation. It may be one of IRD_IMMEDIATE, IRD_BLOCKFOREVER or some integer N indicating: return with immediate failure if resources are not available, block forever (indefinitely) until resources become available, or wait at most N hundsecs for resources respectively. Defines are in irDefines.h. The default is the current value of the TSM parameter RESOURCE_RETURNMODE. RESOURCE_RETURNMODE is set through **setparam**. By default TSM sets RESOURCE_RETURNMODE to 10 seconds.

Return Values

The return value placed in *r.0* as follows:

- 0 — if the resource is successfully allocated
- -1 — if a system error occurs
- -2 — if no resources are in service or
- -3 — if no resources are currently available

See Also

sp_alloc

rts

Name

This script instructions returns from a script subroutine.

Synopsis

rts()

Description

The **rts** instruction is the mechanism for returning from a subroutine call. The saved values for all registers except *r.0* are restored. *r.0* is left at whatever value it was before the **rts** instruction.

Example

In the following example, after speaking the character string in STKSYM with a falling inflection and then the phrase “has not opened,” the script goes to dont_count. The rts in dont_count causes the next instruction to be executed after the subroutine call in not_opened.

```
MAIN:

  ........   .............. ...................
 SR_CALL( )
    ................ ............... ..................

SR_CALL:

    ................  ............... ....................

not_opened:
 tchars(ch.STKSYM,’f’)
 talk("has not opened")
 goto dont_count
.......... ........... ..................

dont_count
  ...........  ..................  ...................
 rts()
```

say

Name

This script instruction plays ASCII text stored in a buffer.

Synopsis

say(*ctype.src*)

Description

The **say** instruction instructs the system to speak ASCII text stored in a buffer. The *ctype.src* argument specifies the ASCII text string to be spoken. The script may pass text as a literally quoted string or the contents of a null-terminated field (for example, previously populated with a call to the **dbase** instruction). The maximum length of a literal string is 2048 characters.

Say is similar to the **talk** instruction used for phrases of coded speech. The text passed to say is stored in a buffer that will hold up to 2048 bytes of text. This buffer is flushed and the text is played when the buffer is full and another **say** instruction is executed or when any wait-causing instruction is executed.

The **tflush** instruction may be used to flush the TTS buffer and cause the text to play. The first two arguments to tflush (the *must_hear_flag* and the *wait_indicator*) have the same effect for text-to-speech as for coded speech. (The third argument to **tflush**, the *remember_flag*, is not used for TTS.) That is, the first argument may be used to disable “talkoff” and the second may be used to play speech and to continue the script without waiting for the play to complete. Normally, TSM waits for a TTS play to complete before going to the next instruction. “Spinning off” a TTS play, then executing **dbase** to get the next block of text while the first block is playing avoids a delay in play

between the two blocks of text. Scripts may continue executing alternate **say**, **tflush**, and **dbase** calls in this manner until all the text from a DIP is passed to say to be played.

The **say** instruction returns one of following values in script register 0 (*r.0*) (Table 25):

Table 25. Return Values for the say Instruction

Return Value	Return Explanation
0	The **say** instruction completed successfully
-1	The **say** instruction failed. This happens if the text passed to say did not fit into one TTS buffer (2048 bytes).

As with coded speech, any text-to-speech being played stops when the script that caused it terminates or executes a tstop instruction.

Examples

In the following example, the text for both of the say instructions is sent to the SSP circuit card for analysis. Eventually, the text is spoken as synthesized voice.

```
/* Need to append a space to text continued */
/* in next say() instruction */
say(”For everything there is a season, “)
say(”and a time for every purpose under heaven.”)
tflush()
```

In the following example, the say instruction uses the DEALER name obtained in the **dbase** call for spoken output. Make sure that the DEALER is not null or the say instruction has nothing to output.

```
#define DIPmsg 100
#define DEALER 124 /* assuming text is at 24 byte
   /* offset in message */

dbase(DIP, GET_DEALER, ch.DIPMSG, 100, 0, 0)
talk("The name and address of your local dealer is")
say(ch.DEALER)
tflush()
```

See Also **fsay**

scrinst

Name This script instruction determines the number of instances of a script currently running on the system.

Synopsis **scrinst([*ctype.script*])**

Description The **scrinst** instruction enables an application script to find out how many instances of a script are running currently on the system. Based on the value returned by this instruction, the script may choose to prohibit execution of

another instance of the script (via the *exec* instruction) or the script may quit if it is performing a check on itself and has exceeded the limit.

The *ctype.script* optional argument is the script, or service, name. If no script name is given, the script executing the instruction is assumed. This instruction sets the value of register 0 (*r.0*) to the number of instances of the given script at the time the instruction is invoked.

There are several possible uses of *scrinst* based on the ways in which a script may be started:

- Incoming call — It is suggested that the method of limiting the number of scripts started with an incoming call be left as it is. That is, do not assign a service to a number of channels greater than the desired limit. If the number of channels assigned to a script exceeds the limit, a script still may check the instance count as its first task and quit before answering the call if the instances exceed the limit.
- exec — The **exec** script instruction is the primary means by which an instance limit may be exceeded. Therefore, any application script that is concerned about running too many instances of another script should use scrinst for that script before using *exec*.

  In this case, it is important to avoid a wait condition in the interval between **scrinst** and **exec**. This could cause other scripts running simultaneously that are performing the same test to get identical results from **scrinst** before any of them perform the exec instruction. Use **tflush** before **scrinst** to play any speech that is queued. Otherwise, the **exec**

instruction plays the speech and the script waits for the play to complete before performing the **exec** instruction.

- Soft seizure and virtual seizure — Scripts started by a soft seizure or virtual seizure request from a DIP may use **scrinst** to check themselves against an instance limit as their first task, similar to the way **scrinst** may be used if the script is started by an incoming call. If the script determines that it cannot continue, it may signal the DIP that started it by using the *dipterm* instruction and calling quit with a specific value that the DIP may check.

Examples

In the first example, the script requests the number of instances of the script *riverbank* currently running on the system. In the second example, because no argument is given, the script requests the number of instances of itself running on the system.

```
scrinst("riverbank")
scrinst()
```

setalk

Name

This script instruction specifies a new talkfile.

Synopsis

setalk(*type.talk*)

Description

The **setalk** instruction is used to specify a new talkfile. The argument, *type.talk*, is the id of the new talkfile. Do not use a character data type as the *type.talk* argument.

After **setalk** is executed, the previous talkfile id is returned in register 0 and can be saved for future use.

The **setalk** instruction overrides the talkfile number that is contained in the first listfile specified in the **tfile** instruction.

Example

In the following example, the new talkfile is set to talkfile 25. The phrase number 210 spoken by the **talk** instruction refers to the speech phrase encoded in talkfile 25.

```
setalk(25)
talk(210)
```

See Also

setftalk

setattr

Name

This script instruction statically sets an attribute associated with a script.

Synopsis

```
#include “tas_defs.h”
setattr (attribute)
```

Description

The **setattr** instruction statically sets an attribute associated with a script. Several attributes may be combined by several invocations of setattr.

The attributes that **setattr** modifies are static and control functions that take place before a script is started on a channel. It is not possible to vary dynamically a script's behavior that is controlled with **setattr**. Therefore, the **setattr** instruction should not be used to set conflicting attributes (for example, by using both **setattr(ATTR_ANI)** and **setattr(ATTR_SID_0)** instructions).

Valid attributes are:

ATTR_ANIANI only
ATTR_ANI_OANI only
ATTR_ANI_PANI preferred
ATTR_SID_OSID only
ATTR_SID_P SID preferred

Example

For example, to set an attribute that requests a station identification (SID) for the calling party number (CPN), use:

setattr (ATTR_SID_O)

setcca

Name

This script instruction sets the type of Call Classification Analysis (CCA) at the script level.

Synopsis

setcca(*type.mode,type.nrings,type.ansdet)*

Description

The **setcca** script instruction allows application developers to set Full CCA parameters at the script level. The parameters that can be set are:

- *type.mode* — This parameter can be either Intelligent or Full CCA. If mode is 0 (default), Intelligent CCA is used. If mode is 1, Full CCA is used (available in US and Canada).

Note: The *type.nrings* and *type.ansdet* are not used by Intelligent CCA.

- *type.nrings* — Number of rings to wait for answer. This parameter can be between 1–10 rings.
- *type.ansdet* — Answer detection by voice energy detection. 0 = no, 1 = yes. The default is -1 (yes for Tip/Ring and LSE1/LST1 lines, no for T1 (E&M), E1 (CAS), and PRI). By default, answer detection is turned on for Tip/Ring and LSE1/LST1 lines and off for T1 (E&M), E1 (CAS), and PRI lines because Tip/Ring and LSE1/LST1 lines do not have answer supervision while T1 (E&M), E1 (CAS), and PRI lines do. Answer supervision is more reliable in detecting answer than voice energy detection.

Note: If you use Full CCA as the mode, do not use the **tic('W',** *type*.*rings***)** or **tic('w',** *type*.*rings***)** instruction.

Note: For accurate transfer results, assign Full CCA only to an SP circuit card.

Example

In the following example, the call classification parameters are set to Full CCA, ten rings, and answer detection is enabled for Tip/Ring and LSE1/LST1 lines and disabled for T1 (E&M), E1 (CAS), and PRI lines.

```
setcca(1,10,-1)
```

The following example is an excerpt from a script showing how a developer might use the **setcca** and **tic** instructions in a Full CCA application.

```
setcca(1,10,-1)
nextcall:
dbase( .... )     /* get number to dial from DIP */

tic('O', r.3)    /* call number in register 3 */

jmp(r.0 == 'N', noAns)     /* no answer after 10 rings */
jmp(r.0 == 'B', busy)
jmp(r.0 == 'F', retry)
jmp(r.0 == 'A', answer)
jmp(r.0 == 's', SIT)
jmp(r.0 == -4, noResource)

noAns:
tic('h')     /* put line on-hook to stop ringing */
```

```
busy:
dbase ( .... )    /* report result to controlling DIP */
goto (nextcall)

SIT:
jmp(r.1 == 'R', retry)
jmp(r.1 == 'r', retry)
jmp(r.1 == 'K', retry)
jmp(r.1 == 'k', retry)
dbase ( .... )     /* report result to controlling DIP */

answer:
talk("Hello, you may be the winner of a free trip to Hawaii")
dbase ( .... )     /* report result to controlling DIP */
goto (nextcall)
```

setftalk

Name

This script instruction specifies a new prepend directory for the **fsay** and **ftalk** instructions.

Synopsis

setftalk(ctype.src)

Description

The **setftalk** instruction is used to specify a new prepend directory for the fsay and ftalk instructions. The argument, *ctype.src*, is the full pathname of the prepend directory.

The prepend directory cannot be longer than 128 characters.

When a relative pathname is given to **fsay** or **ftalk,** the sum of lengths of the prepend directory and the relative pathname cannot exceed 128 characters.

Example

See Also

fsay
ftalk

setIRAPIparam, setIRAPIparamstr

Name

setIRAPIparam, setIRAPIparamstr, getIRAPIparam, getIRAPIparamstr - set/get the value of a IRAPI library channel-based parameter.

Synopsis

#include <*irapi.h*>
getIRAPIparam (*identifier, type.dest*)
getIRAPIparamstr (*identifier, ctype.dest, type.count*)
setIRAPIparam (*identifier, type.value*)
setIRAPIparamstr (*identifier, ctype.string, type.count*)

Description

The **setIRAPIparam** and **setIRAPIparamstr** functions assign values to an IRAPI library parameter, while **getIRAPIparam** and **getIRAPIparamstr** get the current value of a parameter. See IrPARAMETERS(4IRAPI) for a list of valid parameters and their default and legal values.

getIRAPIparamstr 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, **getIRAPIparamstr** ignores any NULL characters in the parameter data. It also makes no attempt to NULL terminate the *string*. **setIRAPIparamstr** also ignores NULL characters and copies the number of bytes for the parameter [specified in IrPARAMETERS(4IRAPI) or through *count*] into the call profile. **setIRAPIparamstr** copies only *count* bytes of data beginning at the address specified by *value*, therefore, *value* need only point to an area of size *count*.

Parameters are preserved across **exec** boundaries.

Return Values

getIRAPIparam, **getIRAPIparamstr**, **setIRAPIparam**, and **setIRAPIparamstr** return 0 in register 0 (*r.0*) if successful and return -1 in register 0 (*r.0*) if an error occurs.

See Also

IRAPI Manual Pages: **irParams**(3IRAPI), **IrPARAMETERS**(4IRAPI), **irGlobalParam**(3IRAPI)

setparam

Name

This script instruction sets a parameter associated with a script.

Synopsis

include “tas_defs.h”
setparam (*type.param, type.value*)

Description

The **setparam** instruction dynamically sets a parameter associated with a script. Note that Register 0 (*r.0*) is set to a negative number if the instruction fails.

The following are possible parameters that can be specified:

- SERVICE_TYPE — Change the service type for outbound PRI calls. Valid service types are (Table 26):

Table 26. Valid Service Types

Service	Service Type
SDN (including GSDN)	SVC_SDN
MEGACOM 800	SVC_MEGACOM800
MEGACOM	SVC_MEGACOM
INWATS	SVC_INWATS
	1 of 2

Table 26. Valid Service Types

Service	Service Type
WATS maximal subscribed band	SVC_WATS
ACCUNET switch digital	SVC_ACCUNET
Nodal Long Distance Service	SVC_NODAL_LDS
International 800	SVC_I800
ETN	SVC_ETN
Private Line	SVC_PRIVATE_LINE
DIAL IT NOVA, MULTIQUEST	SVC_DIALITNOVA
Reserved (CNO)	SVC_RESERVED_CNO
	2 of 2

Note: If you specify a service type that is not available, typically the tic instruction returns the value ‘p’ in *r.0*. *r.1* typically contains the value CV_MIGGINGIE (96) or CV_SERVICENA (63) depending on the switch software.

If the service you need to specify is not included in tas_defs.h, use the following lines:

```
#define SPECIAL_SERVICE X  /* service type you want to use */
setparam(SERVICE_TYPE, SPECIAL_SERVICE)
```

where *X* is a number between 0 and 31 that is to be placed in the Facility Coding Value field of the Network-Specific Facilities information element of the ISDN SETUP message.

- BEARER_CAP — Change the bearer capability for PRI calls. Valid bearer capabilities are shown in Table 27:

Table 27. Valid Bearer Capabilities

Bearer	Bearer Capability
Speech	BEAR_SPEECH
Unrestricted digital	BEAR_DIGITAL
Restricted digital	BEAR_RDIGITAL
Modem 3.1 KHz audio	BEAR_MODEM

- RESOURCE_RETURNMODE — Change the return resource mode for shared system resources (that is, VOICE, TTS, RECOG, etc.). By default TMS waits up to 10 seconds if it cannot immediately get an SSP resource needed for a script. Valid values are:
  - ~ IRD_BLOCKFOREVER (defined in **irDefines.h**) — Wait as long as necessary for the resource to become available
  - ~ IRD_IMMEDIATE (defined in **irDefines.h**) — Fail if the resource is not immediately available
  - ~ ***<N>*** where *N* is any positive integer — Wait for *N* hundredths of a second for the resource to become available

Example

The following example shows how to use the **setparam** instruction in an application to specify Nodal Long Distance Service when placing an outbound call.

```
#include “tas_defs.h”    /* Contains SERVICE_TYPE macro
definitions */

#define DIALED_NUMBER    0  /* Location of character string to
be dialed */

     /* Specify the speech file you wish to use */
     tfile(application)

Begin:
     /* Set service type for outgoing call to Nodal Long
```

```
Distance */
     setparam(SERVICE_TYPE, SVC_NODAL_LDS)

     /* initialize character string to be dialed */
     strcpy(ch.DIALED_NUMBER, ”6145551212”)

     /* Originate call */
     tic (‘O’, ch.DIALED_NUMBER)

     ...
```

setstring

Name

This script instruction sets string parameter associated with a script.

Synopsis

#include “tas_defs.h”
setstring (*type.src, ctype.dst*)

Description

The **setstring** instruction sets a string parameter associated with a script. For example, **setstring** can be used to set the calling party number on outbound calls by setting the OUTBOUND_ANI parameter. After **setstring** is invoked, subsequent outbound calls use the *ctype.dst* argument as the outbound calling party number.

Register 0 is set to -1 if the *ctype.dst* argument is too long or if the *type.src* argument is invalid. Register 0 is set to -2 if the *ctype.dst* argument is not a valid number.

Examples

The following examples set the calling party number of an outbound call to (614) 555-1212:

Example 1:

```
setstring (OUTBOUND_ANI,”6145551212”)
```

Example 2:

```
load (int.STRINGTYPE OUTBOUND_ANI)
strcpy (ch.CPN ”6145551212”)
setstring (int.STRINGTYPE ch.CPN)
```

setttfl

Name

This script instruction sets touch-tone flushing.

Synopsis

setttfl(***type.flg***)

Description

The **setttfl** instruction allows the script to change the way TSM handles the touch-tone buffer. Normally, TSM gets rid of any touch tones it has received for the script when the speech buffer is flushed and speech is played. The **setttfl** instruction disables the TSM action of clearing the touch-tone buffer whenever speech is played.

If the *type.flg* argument is 1, touch-tone flushing is turned on. If the setttfl instruction is not used, the default condition is to set touch-tone flushing to on.

If *type.flg* is 0, touch-tone flushing is turned off and playing speech does not cause the touch-tone buffer to be cleared. If touch-tone flushing is turned off and talkoff has been enabled on the channel (using the **tflush** instruction with the must_hear_flag set to 0), an instruction that normally plays the queued phrases now clears any phrases queued in the phrase buffer. This happens because phrases that are in the buffer are assumed to be part of the prompt that the talkoff touch tones affect. With talkoff enabled, phrases that are already queued will not be heard. Instead, the script advances to the appropriate point based on the touch-tone input received.

Example

`setttfl(0)` — Turn off touch-tone flushing

`setttfl(1)` — Turn on touch-tone flushing (default)

See Also

getdig
getinput
ttclear

sleep

Name

This script instruction causes the script to sleep.

Synopsis

sleep(*type.src*)

Description

The **sleep** instruction makes the script do nothing for the number of seconds specified by the argument. See the **event** instruction for TSM events that may interrupt the sleep instruction before the specified time has passed.

The **sleep** instruction is a speech flushing instruction and causes any queued speech to be played before the sleep is done

Example

In the following example, the script dials out on a channel, then waits 5 seconds for completion of the dial before continuing.

```
.............. ................ ........
tic('d',int.PHONENBR)
sleep(5)
.........  ............  ............
```

See Also

event
nap

sp_alloc

Name

This script instruction explicitly allocates/deallocates speech recognition resources.

Synopsis

sp_alloc(*type.onoff, type.resource*[,*type.mode*])

Description

Note: The instruction **resource_alloc** replaces **sp_alloc**. Continued use of **sp_alloc** is discouraged. This instruction can only be used with the SSP circuit card. It cannot be used for the LSPS II circuit card.

The **sp_alloc** instruction is explicitly used to allocate and deallocate speech recognition on the SSP circuit card.

The **sp_alloc** instruction may be used by a script to allocate the speech recognition resource on the SSP circuit card. Normally, this resource is shared by all scripts running on the system, and allocation is done automatically only when the script actually uses the resource. If the SSP resource is not available when an instruction that requires it is executed, the instruction fails. By using **sp_alloc**, the script may test for the availability of a particular SSP resource. If the resource is available, it is allocated to the script until the script terminates or until the script deallocates the SSP resource using **sp_alloc**.

sp_alloc may be used to allocate an SSP resource for a period longer than the script is actually recognizing speech. Care should be taken to avoid

overloading the systems SSP facilities, since this can occur if many scripts using **sp_alloc** are running simultaneously. Script register 0 (*r.0*) is set to the following values to indicate the status of the **sp_alloc** execution:

0	Success
-1	Error (sp_alloc already on or off)
-2	System resources not available

The *type.onoff* argument is used to tell **sp_alloc** whether to allocate or deallocate resources. Its two valid values are as follows:

1	Allocate the SSP resource
0	Deallocate the SSP resource

The *type.resource* argument is used to tell the sp_alloc which SSP resource or combination of SSP resources to allocate or deallocate. Each SSP resource has a unique value. The values for each resource and examples of how resources can be added are listed below in the following tables (Table 28 on page 529, and Table 29 on page 530):

Table 28. Resource Values:

1	Voice coding or playing
2	PRI function
4	WholeWord Speech Recognition (SR)
8	Call Classification
16	Text-to-Speech
64	Echo cancelling
256	FlexWord Speech Recognition (SR)

If the *type.onoff* argument is 1, the optional *type.mode* argument may be used with the following values:

- IRD_IMMEDIATE (default as defined in **irDefines.h**) — Allocate resources immediately
- IRD_BLOCKFOREVER (defined in **irDefines.h**) — Wait until resource becomes available before continuing
- <*n*> — Wait <*n*> hundredths of a second for resource to become available before continuing, where *n* is a positive integer

The values for the WholeWord recognition and FlexWord recognition resources can be added together to allocate or deallocate more than one SSP resource by using one **sp_alloc** instruction. See Table 29 for examples.

Table 29. sp_alloc Examples

Action	*sp_alloc Script Instruction*
Allocate WholeWord recognition resource for the script	**sp_alloc(1,4)**
Deallocate FlexWord recognition resource for the script	**sp_alloc(0,256)**
Allocate both WholeWord recognition and FlexWord recognition resource for the script	**sp_alloc(1,260)**

See Also **resource_alloc**

sr_talkoff

Name This script instruction enables/disables speech recognition during prompt.

Synopsis **sr_talkoff(*flag*)**

Description The **sr_talkoff** instruction is used to enable/disable speech recognition during the prompt (barge-in). When speech recognition during prompt is enabled using **sr_talkoff(1)**, the **getinput** instruction starts playing any

phrases in its queue and simultaneously turns on the recognizer. When speech recognition during prompt is disabled with **sr_talkoff(0)**, **getinput** plays any phrases in its queue, then turns on the recognizer. Caller speech or touch-tone input interrupts any of the phrases that were started with the **getinput**. If a **tflush** instruction is used to initiate phrases immediately before the **getinput**, the recognize during prompt does not apply to those phrases (because the recognizer is not on).

If . . .	Then . . .
Speech recognition is enabled with *sr_talkoff(1)*,	**getinput** plays phrases and *simultaneously* turns on the recognizer.
Speech recognition is disabled with *sr_talkoff(0)*,	**getinput** plays phrases *then* turns on the recognizer.
tflush starts playing phrases before **getinput**,	Recognize during prompt does not apply. (Recognizer is not on.)

If recognition during prompt is enabled, the call can be received through a network interface circuit card (E1/T1 or Tip/Ring) that is connected to the TDM bus with the ‘‘tdm’’ option set. Enabling **sr_talkoff** requires that the SSP or LSPS II circuit card play the prompts. The E1/T1 circuit cards are already set to ‘‘tdm.’’ However, for Tip/Ring circuit cards set to ‘‘talk,’’ the system detects a ‘‘recognition during prompt’’ request in the script and automatically uses an SSP or LSPS II circuit card to play the prompts. If a Tip/Ring circuit

card has the ‘‘talk’’ option set and **sr_talkoff** is off, the system plays all prompts with the Tip/Ring circuit card. If a Tip/Ring circuit card has the ‘‘tdm’’ option set or **sr_talkoff** is on, the system plays all prompts with the SSP or LSPS II circuit card.

Because playing prompts uses resources on the SSP or LSPS II circuit card, application designers who find that the SSP or LSPS II resources are strained may want to consider configuring their Tip/Ring circuit cards with “talk” and designing the application to use as few speech processor resources as possible.

Example

In this example, speech recognition is enabled during the prompt.

```
sr_talkoff(1)
```

strcmp

Name

This script instruction compares two character strings.

Synopsis

strcmp(*ctype.src,ctype.src*[,*type.len*])

Description

The **strcmp** instruction compares two character strings and returns the result of the comparison in register 0 (that is, *r.0*). The return value is interpreted as follows:

If *r.0* is:

=0	The strings are equal
<0	The first string is lexicographically less than the second string
>0	The first string is lexicographically greater than the second string

The c*type.src* argument can be either an address or a literal string. If the optional *type.len* argument is used, the comparison is limited to the number of characters specified by the argument.

Examples

In the following example, **strcmp** compares the literal string XYZ?:136 to the string in location char.20. If they are equal (exactly the same) the script jumps to the label equal.

```
strcmp ("XYZ?:136",char.20)
jmp(r.0 ==0,equal)
```

In the following example, the string stored at location int.56 is compared to the string located at char.80. If the first string is greater or equal to the second (that is, would be listed after the string at char.80 in an alphabetical listing or is exactly the same as char.80), the script jumps to the label “greg.”

```
strcmp(int.56,char.80)
jmp(r.0 >= 0,greg)
```

strcpy

Name

This script instruction copies the source string to the destination.

Synopsis

strcpy(*ctype.dst,ctype.src*[,*type.len*])

Description

The **strcpy** instruction copies a character string specified by the *ctype.src* argument to the address specified by the *ctype.dst* argument. The *ctype.dst* argument must be a string address. The *ctype.src* argument can be an address or literal string.

If the optional *type.len* argument is used, the instruction will copy, at most, the number of characters specified by that argument. The result may or may not be null terminated, depending on whether a null terminated character was copied before the *type.len* character limit was reached.

Examples

In the first example, the literal string ABCDEFGHI is copied into char.10. In the second example, the string stored in char.10 is copied to char.50.

```
strcpy(char.10,"ABCDEFGHI")
strcpy(char.50,char.10)
```

strlen

Name

This script instruction computes the length of a string.

Synopsis

strlen(***ctype.src***)

Description

The **strlen** instruction computes the length of the string specified by the *type.src* argument. The *type.src* argument can be a literal string or the location of a string. The length of the string (that is, number of characters in the string) is returned in register 0 (*r.0*).

Examples

In the following example, the length of the string is stored at location char.19. If the length of the string is less than 10 characters, the script jumps to the label Lab1.

```
strlen(char.19)
jmp(r.0 < 10,Lab1)
```

In the following example, the length of the literal string AB123,:=+ is computed (9 in this case) and stored in register 0. Since *r.0* contains 9, the script will not jump to Lab2).

```
strlen("AB123,:=+")
jmp(r.0 !=9,Lab2)
```

subprog

Name

This script instruction executes a TSM script or IRAPI application as a subprogram, and then returns to the parent script with data.

Synopsis

subprog(*ctype.appname*[, *type.data, type.nbytes*])

Description

The **subprog()** instruction is similar to the **exec** instruction. It may be used to run another TSM script or IRAPI application. When an application is run with **subprog()**, however, the calling script (parent) is not replaced by the new application. Instead it is run as a subprogram and returns to the point where **subprog()** was called when the called application terminates. Data may also be passed in both directions via **subprog()**.

Note: Use of **subprog()** involves a UnixWare process level context switch that may decrease system performance if used excessively. Do not use **subprog()** if you can use a script subroutine to do the same thing. See **label()** instruction.

TSM scripts that are run with **subprog()** have direct access to a parent script's data via the special "&ch.", "&int.", and "&short." data type specifiers. When one of these data types is used, they must be followed by a TSM register number containing an offset into the parent script's data space which references the desired data. The name of the parent script may be obtained with the special "&script.0" argument. The value of this argument will be a null string if there is no parent script.

Note: If the parent is an IRAPI application that calls a TSM script with **irSubProg**, this method cannot be used to access data in that application. Using the '&' data types will always access the nearest ancestor TSM script (if any). For example, if TSM script A **subprog()**s IRAPI application B, and B in turn **irSubProg()**s TSM script application C, then A is the nearest ancestor TSM script to C. In this case, when using the '&' argument data types, C will access A's data, not B's. If there is no ancestor TSM script to access, the instruction using these argument types will fail. It's best to check the "&script.0" value first to see if there is a parent script to access and get its name. Note that a running script can get its own name through the "script.0" argument.

The *ctype.appname* argument is the name of the application to be executed.

The optional *type.data* and *type.nbytes* arguments are used to pass a block of data to the new application. The *type.data* argument specifies the location of the data and *type.nbytes* specifies the size of the data in bytes. If *type.data*

is a register or immediate, *type.nbytes* is ignored and the size of an integer (4 bytes) is assumed.

The data passed to the new application via the *type.data* argument is available to a TSM script via the special X.0 argument, just as if the *exec* instruction were used to pass the data. IRAPI applications have access to this data through the IRP_EXEC_BUF parameter. If the called application modifies X.0 (or IRP_EXEC_BUF), that modification will be accessible by the calling application when **subprog()** returns.

All TSM register values are also passed to the new application. Script applications access these values in their own registers (r.REGNUM argument type). IRAPI applications may access these values through the IRP_REGISTER parameter.

Note: The subprogram stack is limited by the UnixWare system tunable parameter NCONTEXTSTACK, which is set to three by default. This limits the use of **subprog()** to three levels per application unless you tune NCONTEXTSTSACK to a higher value (limit of 10) by using the UnixWare idtune command. See the UnixWare books details.

Return Values

subprog() returns values in both register 0 (*r.0*) and register 1 (*r.1*). *r.0* is 0 if **subprog()** was successful. It contains a negative value to indicate failure. If successful, **subprog()** also returns in *r.1* whatever value was passed to the **quit()** instruction by a called TSM script application, or whatever value was passed to **irReturn** by a called IRAPI application.

Examples

The following example executes an application called "APPL", passing it 100 bytes of data and checking return values:

```
subprog("APPL", ch.DATA, 100)
jmp(r.0 < 0, L_Failure)
quit(r.1)/* quit with return value from APPL */ L_Failure:
talk("failed to execute application")
quit(-1)
```

See Also

exec(),
execu()
label()
quit()

IRAPI Manual Pages: **irSubProg**(3IRAPI), **irReturn**(3IRAPI), **IrPARAMETERS**(4IRAPI)

talk

Name

This script instruction speaks one or more phrases.

Synopsis

talk(type.src*[,type.style]*)

Note: The ability for the talk instruction to specify multiple phases using commas to separate arguments is no longer supported.

Description

The **talk** instruction uses the *type.src* argument to specify the phrase to be spoken. The second optional argument is the coding style of the speech to play back. For the LSPS II circuit card, all speech to be played in one group of requests must use the same coding style.

Note: Do not use character data types for arguments in either format.

When **talk** (as well as **tchars** and **tnum**) instructions are executed, the system queues phrases in a buffer, but the phrases are not immediately played. Phrases are played under one of the following three conditions:

1. The script executes a speech-flushing instruction
2. The script executes a **say** instruction (Text-to-Speech)
3. The number of queued talk or say requests exceeds the TSM limit in the **/vs/data/irAPI.rc** file. If there is not limit, the default of 256 is used. When the LSPS II circuit card is used for speech playback or text-to-speech, the limit is 64.

Example

In the following example, the GREET subroutine is called on to say two introductory phrases from the talkfile specified in the **setalk** instruction. It then calls on a subroutine to speak the final phrase contained.

```
MAIN:
 setalk(30)
 GREET( )
 ..... ..... .....
 BYE( )
```

```
GREET:
 talk(10)
talk(11)
 indirect_talk(50)
............. ............... ........
 rts( )
indirect_talk:
 talk(r.3)
 rts( )
```

See Also

ftalk
setalk
tchars
tnum

talkresume

Note: This instruction is only available for speech playback the SSP circuit card, *not* for the LSPS II circuit card.

Name

This script instruction starts playing queued phrases at a specified point.

Synopsis

talkresume(*type.offset*)

Description

The **talkresume** instruction plays whatever phrases remain from the last **tflush** instruction starting at the point they were interrupted (that is, by talk off) plus the given offset in seconds. If the offset is a positive number, speech is played from a point after the interruption. If the offset is a negative number, speech is played from a point before the interruption. If the offset is 0, play starts at the point where the interruption occurred. If VROP has played all of the phrases, only a negative offset has any effect.

The **talkresume** instruction stores a return value in register 0. If the value is negative, an error has occurred. If the value is 0, play completed successfully. If the value is +1, the “play complete” was caused by talkoff. If the value is +2, there was no speech left to play (that is, **talkresume** was given with a non-negative offset when VROP had already played all the speech).

For **talkresume** to work properly, the speech it affects must have been played originally with the **tflush** instruction with the optional *remember_flag* argument set to 1. This tells VROP to “remember” the speech that tflush tells it to play and to keep track of where that speech is interrupted. Subsequent calls to talkresume then have the desired effect on this speech. VROP remembers the speech it was playing until it receives another set of phrases to play by subsequent script instructions. Only one set of phrases can be remembered per channel at a time.

Example

In the following example, the script is instructed to skip ahead four seconds, then resume talking.

```
talkresume(4)
```

tchars

Name

This script instruction speaks phrases from a variable name.

Synopsis

tchars(*ctype.src{*[,***'inflection'***],*[*type.style*]*}*)**

Description

The **tchars** instruction puts the null terminated string of alphanumeric characters that are identified by the first argument into a queue for speaking. The second argument, when specified, controls the speech inflection. The three inflection parameters are r (rising), f (falling), and t (total). Total produces both a rising inflection on the first phrase and a falling inflection on the last phrase if there is more than one; it produces a falling inflection if there is only one phrase. It is important to note that “r”, “f”, and “t” work only if those types of phrases are in the talkfile.

The optional *type.style* argument specifies the coding style.

The **tchars** instruction speaks one character at a time, unlike the tnum instruction, which speaks the digits as one number. For example, the **tchars** instruction would speak the number 61 as “six-one” while the **tnum** instruction would speak “sixty-one.” Also, tchars speaks the string “61A” as “six-one-A.”

Example

In the following example, the script asks the caller to enter his/her ID number. The script waits for three touch tones, which it stores in ch.ID. The script then repeats what the caller entered, reading the value in ch.ID.

```
talk("Please enter your ID number")
getinput(ch.ID,3)
talk("Your ID number is")
tchars(ch.ID)
tflush()
```

See Also

talk
tflush
tnum

tfile

Name

This script instruction identifies a talkfile.

Synopsis

tfile(“*application_name*.pl”[,”*talkfile 2*”...])

Description

The **tfile** instruction indicates the speech database to use for the script. The first listfile name, called *application_name*.pl (see Chapter 2, Application Structure), is the name of the primary listfile. Its talkfile number is used for the initial setalk and is used for **tnum**, **tchar**, and **talk** instructions if the **tfile** portion of the phrase ID is 0.

Each phrase in the talkfile is identified by a unique number and string in the listfile. Because the TAS uses this information, the **tfile** instruction must be specified in the script before the first voice output instruction.

Phrases in the primary listfile are not bound to the talkfile when the script is compiled. They are played from the talkfile currently in effect when the **talk** instruction is executed. However, any additional listfiles given in the **tfile** instruction have the talkfile and phrase number bound when the script is compiled. Phrases selected from these listfiles are not affected by changes in the talkfile that occur during script execution.

Example

In the following example, the **tfile** instruction specifies the file of application phrases that are accessed by the voice output instructions, where the *applN* identifier in the file name representing the *application-name*.

```
MAIN:
 tfile("/speech/talk/STOCKS.pl")
 GREET( )
.......... ........... ............
 BYE( )
```

See Also

setalk
talk
tchars
tnum

tflush

Name

This script instruction outputs the speech buffer unconditionally.

Synopsis

tflush([*must_hear_flag*][,*wait_indicator*][,*remember_flag*])

Description

Phrases specified by a script to be spoken with the **talk**, **ftalk**, **tchars**, or **tnum** instruction are queued (that is, not spoken) pending the encounter of a **tflush** instruction or a data gathering instruction.

The accepted values for the arguments are:

must_hear_flag	0	Touch tones entered during play or voice coding cause play or voice coding to stop (default)
	1	Touch tones entered during play or voice coding do not cause play or voice coding to stop
wait_indicator	0	Wait for the play to complete before continuing script execution (default)
	1	Do not wait for the play to complete. Continue script execution.
remember_flag	1	Remember phrases played by this instruction so they may be played again with the **talkresume** instruction.
	0	Do not remember the speech

The *must_hear_flag* option, when set to a non-zero value, disables talkoff so that any speech activity (voice play or voice coding) on the current channel will not be stopped by touch tones. When using this option with speech play-related instructions (**talk**, **ftalk**, **tnum**, or **tchars**), **tflush(1)** should follow those instructions. When using this option with voice coding (**vc**), **tflush(1)** should precede the **vc** instruction. The talkoff is enabled automatically by the next wait-causing instruction in the script.

The **tflush** instruction returns a value in register 0. If that value is negative, an error occurred. If that value is +1, ‘play complete” occurred because of talkoff. If the value is 0, play has completed successfully.

Examples

In the following example, the script wants the caller to hear music while it processes the transaction with the host computer. After this processing completes, the music is stopped, and the caller is informed of the results and asked to continue entering commands. The tflush instruction does not remember the phrases played by the instruction.

```
  ................................................

talk(int.MUSIC) /* Play music to the caller */
tflush(1,1,0) /* Do not let touch tones turn off music and don’t
wait*/
dbase(0,FUDB,ch.ACCOUNT_ID,8,int.SELL_PRICE, 4) /* Get info
from host */

tstop(1)
talk("Your account has now been credited with Lucent
Technologies stock for the price of")
tnum(int.SELL_PRICE)
talk("Enter your next instruction")
getinput(ch.DIG,2)

.............  ........................
```

In the following example, any touch tones entered are encoded along with the speech.

```
.......... ....................................
tflush(1) /*do not end coding if user enters touch tones*/
vc('b',10,ADPCM32)
```

See Also

talk
tstop

tic

Name

This script instruction controls a telephone interface line (channel).

Synopsis

tic('C', *ctype.dialstr, type.rings*)

tic('D', *ctype.dialstr*)

tic('F')

tic('O', *ctype.dialstr*)

tic('W', *type.rings*)

tic('a')

tic('d', *ctype.dialstr*)

tic('f')

tic('h')

tic('o', *ctype.dialstr)*

tic('w', *type.rings*)

Description

The **tic** instruction provides the script with control functions for the telephone interface line (channel) that the script is currently using. The function that the **tic** instruction performs depends on the value of its first argument. These argument values and their corresponding functions are listed below.

Note: This instruction handles differences between telephony types better than **tic('O')**, or the combination of **tic('o')** and **tic('W')**.

The **tic** instruction uses script registers 0 (*r.0*) and 1 (*r.1*) to return a result. This result may differ according to whether the script is using a Tip/Ring (T/R), T1 (E&M), E1 (CAS), PRI, or LSE1/LST1 channel. Where such variations exist, they are noted below.

- *C* — Call a number and wait for the disposition. Dial *ctype.dialstr*; turn on speech energy detection and wait for number of rings given in type.rings for "answer" (speech energy or ringing stopped), or call progress tone other than ringing, or "no answer."
- *D* — Dial *ctype.dialstr*; wait for any call progress tone, then resume the script.
- *F* — Flash; wait for any call progress tone, then resume the script.

- *O* — Originate (go off-hook and dial ctype.dialstr); wait for the first call progress tone (CPT), then resume the script. Note that without Full CCA, the first CPT could be a Ringback, with no indication of Answer or No answer disposition.
- *W* — Turn on speech energy detection and wait for number of rings given in type.rings for “answer” (speech energy or ringing stopped) or “no answer.”
- *a* — Answer the line (go off-hook).
- *d* — Dial *ctype.dialstr*, then resume the script.
- *f* — Flash the hook (transfer to another line), then resume the script.
- *h* — Hang up the line (go on-hook).
- *o* — Originate (go off-hook and dial ctype.dialstr), then resume the script.
- *w* — Wait for the number of rings given in type.rings for “answer” (ringing stopped) or “no answer”

Table 30 on page 552 lists the possible return values for the different forms of the **tic** instruction. Note that the set of possible return values depends on the type of channel: Tip/Ring, T1 (E&M), E1 (CAS), PRI, or LSE1/LST1.

Table 30. Return Code Results for the tic Instruction

Meaning	Return Values		For tic	Available On			
	r.0	*r.1*		T/R	T1 (E&M) or E1 (CAS)	PRI	LSE1 LST1
Instruction successfully completed	0	•	'a','d','f'[1],'h','o'	•	•	•	•
Answer detected (e.g., voice energy detected or ringing stopped)	'A'	•	'W','w'[2],'C'	•			•[3]
Answer supervision from switch (or DTMF connection tone detected from DEFINITY ECS)	'P'	•	'D','O','W','w','C'	[4]	•	•	4
Busy[5]	'B'	•	'F'[1],'D','O','W','w','C'	•		•	•[3]
Fast busy[5] (reorder tone)	'F'	•	'F'[1],'D','O','W','w','C'	•		•	•[3]

1 of 3

Table 30. Return Code Results for the tic Instruction

Meaning	Return Values		For tic	Available On			
	r.0	*r.1*		T/R	T1 (E&M) or E1 (CAS)	PRI	LSE1 LST1
Ring no answer	'N'	•	'W','w'	•			•[3]
Audible ringing	'R'	•	'F','D','O'	•			•[3]
Dialtone detected[5]	'D'	[6]	'F','D','O','W','w','C'	•		•[6]	•[3]
Stutter dialtone detected	'S'	•	'F'[1],'D','O','W','w', 'C'	•			•[3]
ISDN vacant code[6]	'v'		(any)			•[6]	
Provisioning or protocol error	'p'	6	(any)			•[6]	
Internal hardware or software error or dialing error	-1	•	(all)	•	•	•	•

2 of 3

Table 30. Return Code Results for the tic Instruction

Meaning	Return Values		For tic	Available On			
	r.0	*r.1*		T/R	T1 (E&M) or E1 (CAS)	PRI	LSE1 LST1
Timeout (no call progress tones detected within the timeout period)	-2	•	(all except 'h')	•	•	•	•
Illegal dial string passed[7]	-3	6	'D','O','d','o','C'	•	•	•[6]	•
Touch tone entry detected	't'	•	'O', 'C'	•	•	•	•
Intercept tone heard representing an invalid extension (on DEFINITY ECS or other Lucent Technologies PBX)	's'	'I'	'O', 'D', 'C', 'W', 'w'	•			•[3]
Caller disconnected during transfer	'h'	•	'D','W','w'	•[8]			•[8]

3 of 3

1.A **tic('F')** or **tic('f')** instruction on a PRI channel will always fail (*r.0* = -1).

2.A return value of 'A' in response to a **tic('w')** means only that ringing has stopped before the given number of rings. The speech energy detector is turned on only when a **tic('W')** or **tic('C')** is done.

3.Speech energy detection and Call Progress Tone detection are not available for LST1 on an AYC11 or AYC3B, and timeout (-2) is the most likely value for *r.0*. However, these features are available for LSE1 and LST1 on the AYC21.

4.When connected to a Lucent Technologies DEFINITY ECS or compatible switch that is properly administered for the optional feature Sending DTMF Feedback Tones to the VRU, the application will get an *r.0* value of 'p' for Tip/Ring, LSE1, or LST1 when the connection tone is received. Otherwise, the 'p' is not expected for these types of channels.

5.For PRI channels, the voice system converts information provided by the switch into busy, fast busy, or dialtone call dispositions. An audible tone may not be present.

6.This disposition is not always provided by the switch, the only source of PRI information (without Full CCA). However, when it is provided, more specific info (the ISDN cause value) is available in register 1 (*r.1*). See Table 33 on page 560 for the list of ISDN Cause Values. The **tic** instruction also will return a value in *r.1* when the Full CCA feature is used. See Table 34 on page 566 for a list of *r.1* return values.

7.On an E1 or T1 channel, any dial string with a character other than 1,2,3,4,5,6,7,8,9,0,#,*,A,B,C, D,a,b,c, or d is illegal. PRI channels allow all of the above characters except * and #. On Tip/Ring channels, any string with a character other than 1,2,3,4,5,6,7,8,9,0,#,*,A,B,C,D, a,b,c,d,(,), or – is illegal for touch-tone dialing. For dial pulse dialing on a Tip/Ring channel, any string with a character other than 1,2,3,4,5,6,7,8,9,0,(,), or – is illegal. For E1, T1, and PRI, the maximum dial string is 15 digits. For Tip/Ring channels, the maximum dial string is 30 digits. If more digits need to be dialed than are allowed, multiple **tic('o')**, **tic('O')**, **tic('d')**, and/or **tic('D')** instructions may be required.

8.To get the caller disconnected value ('h' for *r.0*), the Tip/Ring, LSE1, or LST1 channel must be connected to a Lucent Technologies DEFINITY ECS or compatible switch with the optional feature Sending DTMF Feedback Tones to the VRU properly administered on the switch and on the VRU.

If your script encounters a secondary dial tone, you can use a sequence of two **tic** instructions, the first dialing the access number and waiting for secondary dial tone, and the second dialing the remainder of the telephone number.

Due to the characteristics of most switches, you should not require the system to send a flash immediately after answer. The switch may not be prepared to recognize the flash so soon after it detects answer. If your application requires a flash (transfer) after answer, a delay of 1–2 seconds after answer is recommended before sending the flash signal. A short message can be played in this interval to make the delay less noticeable to callers.

Using a sleep or nap instruction after **tic('f')** or **tic('F')** causes the system to disconnect because it detects dial tone and assumes the caller has hung up. The event instruction may be used by a script with the EHANGUP event type to catch this event and prevent disconnect.

If you use the **tic('d')** instruction to send touch tones between two scripts, the tones may be lost if one script sends tones before the other script turns on its DTMF receiver. To avoid this problem, insert a delay of more than 70 milliseconds (for example, use '**nap(10)**' to cause the script to sleep for 100 milliseconds) before the **tic('d')** instruction.

If the system encounters a *glare* condition (that is, an incoming call is detected at almost the same moment the system is dialing out), the system drops the outgoing call and answers the incoming call. The termination of the

script dialing out is treated as a hangup, meaning that if there is an EHANGUP event subroutine defined by the script (see the event instruction), it is executed before the script ends. This may mean that more than the usual number of rings are heard by the caller before the incoming call is answered.

For the **tic('W')** and **tic('w')** instructions, the *type.rings* argument specifies the timeout setting for the instruction, that is, how many rings the system should wait before determining that the call is not answered. TSM sets the timeout value based on the number of rings specified in this field. If the number of rings is greater than 6 and the script does not explicitly set the timeout value (see the **nwitime** instruction), the timeout value is set in the following manner:

timeoutValue (in seconds) = ((nRingsToWait+1)*6) + 5

For example, if you set type.rings to 10, TSM waits until after 10 rings have passed before timing out on the **tic** instruction. If the number of rings is less than 6, the default of 45 seconds is used as the timeout value.

Example

The following example is a portion of a script that uses the **tic** instruction. In this example, the script copies “9999” into NUMBER, then originates a call to that number. Depending on what is returned, the script either jumps to the *end* or *ok* label.

```
#define NUMBER 5

strcpy(ch.NUMBER, "9999")
tic('O', ch.NUMBER)
jmp(r.0 == -1, end)/* hardware failure */
jmp(r.0 == -2, end)/* timeout, no response */
jmp(r.0 == 'B', end)/* busy */
jmp(r.0 == 'R', ok)/* ring */
end:
   quit()
ok:
   tic('h')
   rts()
```

Feature Related Changes (PRI and/or Full CCA)

The following information applies to the **tic** instruction when using it with PRI and/or Full CCA in an application.

Using tic with PRI

The supported **tic** instruction options for PRI are listed in Table 31 on page 559. These options are used in the same manner for PRI as for T1 (E&M).

Table 31. tic Options Supported for the PRI

Option	Function
a	Answer an incoming call
h	Disconnect (hangup) a call
o	Originate a call
C, O	Originate a call & wait for Answer Supervision
d	Dial touch-tone digits

Some options to the **tic** instruction are not applicable to the PRI. These options are listed in Table 32 on page 559.

Table 32. tic Options Not Applicable to the PRI

Option	Function
f or F	Switch Hook Flash
w or W	Wait for Speech Detection
D	Dial digits and wait for tones

The tic('C') and tic('O') Return Values and ISDN Cause Values

The PRI implementation of the **tic('C')** (Call) and **tic('O')** (Originate) instructions provide additional return code information for disconnected calls,

over the T1 (E&M) and Analog interface implementations. Register *r.1* returns the ISDN cause value (if available) for an incomplete call. These cause values are returned by the network and are passed to the script. The cause value is also passed in register *r.1* for a disconnect event. Table 33 contains a list of ISDN cause values returned in register *r.1*. These are arranged in groups according to the more general call disposition value returned in register *r.0*.

Note: If the called party is not an ISDN subscriber, there is no ISDN cause value and -2 is returned to the script. Until ISDN is more widely used, -2 will be a common return. The include (header) file (**/att/include/tas_defs.h**) provides macro definitions of these values. This file can be used by your application by including the following line in your script source file:

#include “tas_defs.h”

Table 33. tic (‘C’) and tic (‘O’) Return Values and ISDN Cause Values for PRI

Call Disposition Value (r.0)	Return Value (r.1)	Meaning
Vacant Code (‘v’)	CV_UNASSNUM (1) CV_ NUMCHANGE (22)	Unassigned number Number changed

1 of 3

Table 33. tic ('C') and tic ('O') Return Values and ISDN Cause Values for PRI

Call Disposition Value (r.0)	Return Value (r.1)	Meaning
Provisioning or Protocol Error ('p')	CV_UNACCEPTCHAN (6) CV_	Channel unacceptable
	NOUSER (18)	No user response
	CV_FACREJECT (29)	Facility rejected
	CV_STATRESP (30)	Status enquiry
	CV_NORMALUNSP (31)	Normal, unspecified
	CV_TEMPFAIL (41)	Temporary resource failure
	CV_FACNOTSUB (50)	Facility not subscribed
	CV_OUTBARRED (52)	Outgoing calls barred
	CV_INBARRED (54)	Incoming calls barred
	CV_BEARERNA (58)	Bearer not available
	CV_SERVICENA (63)	Service not available
	CV_BEARERNI (65)	Bearer not implemented
	CV_CHANNELNI (66)	Channel not implemented
	CV_FACILITYNI (69)	Facility not implemented
	CV_INVALIDCALL (81)	Invalid call reference
	CV_NOCHANNEL (82)	Nonexistent channel
	CV_BADDEST (88)	Incompatible destination
	CV_MISSINGIE (96)	Info element missing
	CV_BADMESS (97)	Nonexistent message type
	CV_BADSTATE (98)	Incompatible message
	CV_INVALIDIE (100)	Invalid info element
	CV_TIMEOUTREC (102)	Recovery on timer
	CV_INTERWORKING (127)	Interworking, unspecified

2 of 3

Table 33. tic ('C') and tic ('O') Return Values and ISDN Cause Values for PRI

Call Disposition Value (r.0)	Return Value (r.1)	Meaning
Dialtone Detected ("D')	CV_NORMALCLR (16)	Normal clearing
Busy ("B')	CV_USERBUSY (17)	User busy
Fast Busy ('F')	CV_NOROUTE (2) CV_CALLREJECT (21) CV_NOCIRCUIT (34) CV_NETWORKDOWN (38) CV_SWITCHBUSY (42) CV_USERIETOSS (43) CV_CIRCUITNA (44) CV_CALLPREEMPTED (45)	No route Call rejected No circuit Network out of order Switching congestion Access info discarded Circuit not available Pre-empted
Answer Supervision ('P')		
Hardware Failure, Undetermined Reason (-1)		
Timeout - No Answer, Reason Not Provided (-2)		
Illegal Dial String (-3)	CV_INVALIDNUM (28)	Invalid Number

3 of 3

You should be aware of the following issues when using these ISDN values:

- Disconnection detected (‘D’) indicates disconnection. It is comparable to dial tone detection on analog lines.
- Busy (‘B’) is comparable to busy, although there may be no audible busy tone.
- Fast busy (‘F’) is comparable to fast busy, although there may be no audible fast busy tone.

Example of PRI Heading

Figure 31 provides an example of how to use this feature in an outbound call script. This script outdials customer numbers retrieved from an account database. If the number dialed was unassigned, invalid, or incomplete, the script sends a message back to the database indicating this. The customer record can then be checked.

Figure 31. PRI Application Using the tic Instruction

```
#include “tas_defs.h”                    /* VIS provide header
file */
#include “dip_defs.h”                     /* Application dip
header file */
#define DIALED_NUMBER 0        /* location of dialed number
string */
               /* Specify the speech file you wish to use */
               tfile(application)
```

```
Begin:
     /* Get the number to be dialed from a database */
     dbase (DIP14, RETRIEVE_NUMBER, ch.DIALED_NUMBER, ... )
     /* Telephone number to be dialed is now in ch.DIAL_NUMBER
*/
     tic('O', ch.DIALED_NUMBER)
     /* Check to see if the call was answered */
     jmp (r.0 == 'P', Continue)/* Call answered, speak */
     /* Call was not answered */
     /* Did PRI indicate that the number does not exist? */
     jmp (r.1 == CV_UNASSNUM, NumberUnassigned)
     /* Did PRI indicate that the number was incomplete? */
     jmp (r.1 == CV_INVALIDNUM, NumberIncomplete)
     /* Did PRI indicate that the number has been changed? */
     jmp (r.1 == CV_NUMCHANGE, NumberChanged)
     /* Otherwise hangup */
     tic('h')
     quit()
NumberChanged:
    /*Send request to database to mark this telephone number as
*/
    /* changed. (Client's number can be updated later) */
     dbase (DIP14, CHANGED_NUMBER, ch.RESULT, ch.DIALED_NUMBER,
... )
     quit()
NumberUnassigned:
     /* Send request to database to mark this telephone number
     /* as unassigned. (Check Client's number for accuracy) */
     dbase (DIP1, UNASSIGNED_NUMBER, ch.RESULT, ch.DIALED_
NUMBER, ... )
```

```
     quit()
NumberIncomplete:
     /* Send request to database to mark this phone number as */
     /* bad. (Client's number can be check for completeness) */
     dbase (DIP1, INCOMPLETE_NUMBER, ch.RESULT, ch.DIALED_
NUMBER, ... )
     quit()
Continue:
     /* continue with normal call processing */
     talk ( "Hello" )
     ...
          quit()
```

Using tic with Full CCA

Be aware that if Full CCA (available only for US and Canada) determines that an outbound call cannot be completed because of a ring-no answer, the transaction should hang up the call using a **tic('h')** as soon as possible. If the call is not hung up immediately, the called party could answer (their phone still is ringing). The application will be unaware of this and will hang up on the called party as soon as the application completes. This not only annoys the called party but also could result in the calling party being billed for a failed call.

Do not use the **tic('W')** or **tic('w')** instruction with Full CCA.

For accurate transfer results, assign Full CCA only to an SP circuit card.

Full CCA Call Dispositions

Table 34 lists the possible return values for the **tic('C')**, **tic('D')**, and **tic('O')** instructions when Full_CCA is turned on via the **setcca** instruction. Note that the set of possible return values depends on the type of channel: Tip/Ring, T1, PRI, or LST1.

Special Information Tones

The special information tones (SITs) shown in Table 34 are returned to the script in register *r.1* when register *r.0* is set to 's'. SITs are available only when Full CCA is used.

Table 34. Call Dispositions for tic ('C'), tic ('D'), and tic ('O') with Full CCA

Meaning	TSM Level		Available On:			
	r.0	*r.1*	T/R	T1 (E&M)	PRI	LST1
Answer detected (for example, voice energy detected)	'A'	0	•	[1]	1	•
Answer supervision from switch (or DTMF connection tone detected from Lucent Technologies DEFINITY ECS or compatible switch)	'P'	0	[2]	•	•	2
Busy	'B'	0	•	•	•	•

Table 34. Call Dispositions for tic ('C'), tic ('D'), and tic ('O') with Full CCA

Meaning	TSM Level		Available On:			
	r.0	*r.1*	T/R	T1 (E&M)	PRI	LST1
Fast busy	'F'	0	•	•	•	•
Ring no answer	'N'	0	•	•	•	•
High and dry	'H'	0	•	•	•	•
Modem tone	'T'	0	•	•	•	•
Dialtone detected[3]	'D'	0[3]	•		•	[4]
Stutter dialtone detected	'S'	0	•			4
ISDN vacant code	'v'	3			•	
Provisioning or protocol error	'p'	3			•	
Internal hardware or software error or dialing error	-1	0	•	•	•	•
Timeout (no call progress tones detected within the timeout period)	-2	0	•	•	•	•
Illegal dial string passed[5]	-3	0	•	•	•	•
CCA resource used up	-4	0	•	•	•	•

2 of 4

Table 34. Call Dispositions for tic ('C'), tic ('D'), and tic ('O') with Full CCA

Meaning	TSM Level		Available On:			
	r.0	*r.1*	T/R	T1 (E&M)	PRI	LST1
Reorder, intraLATA SIT	's'	'R'	•	•	•	•
Reorder, interLATA SIT	's'	'r'	•	•	•	•
No circuit, intraLATA SIT	's'	'K'	•	•	•	•
No circuit, interLATA SIT	's'	'k'	•	•	•	•
Vacant code SIT	's'	'V'	•	•	•	•
Intercept SIT	's'	'I'	•	•	•	•
Ineffective other SIT	's'	'O'	•	•	•	•
Domestic other SIT	's'	'd'	•	•	•	•
International other SIT	's'	'o'	•	•	•	•
International no circuit SIT	's'	'c'	•	•	•	•
International foreign fail SIT	's'	'f'	•	•	•	•

3 of 4

Table 34. Call Dispositions for tic ('C'), tic ('D'), and tic ('O') with Full CCA

Meaning	TSM Level		Available On:			
	r.0	*r.1*	T/R	T1 (E&M)	PRI	LST1
Unknown SIT	's'	'U'	•	•	•	•
Touch tone entry detected	't'	0	•	•	•	•
Caller disconnected during transfer	'h'	0	•[6]	•	•	•[6]

4 of 4

1. By default, speech energy detection is disabled for T1 and PRI channels. However, it can be enabled using the **setcca** script instruction.
2. When connected to a Lucent Technologies DEFINITY ECS or compatible switch that is properly administered for the optional feature Sending DTMF Feedback Tones to the VRU, the application will get an *r.0* value of 'p' for Tip/Ring, LSE1, or LST1 when the connection tone is received. Otherwise, the 'p' is not expected for these types of channels.
3. The disposition of calls on PRI channels is based not only on Full CCA but also on information provided by the switch (the system will respond to the first disposition returned from either source). When this disposition is provided by the switch, more specific information (the ISDN cause value) is available in register 1 *(r.1)*. See Table on page 560 for the list of ISDN Cause Values. If this disposition is provided by Full CCA, register 1 always contains zero, unless *r.0* is 's.'
4. Dialtone and stutter dialtone will not be reported for LSE1 or LST1 when Full CCA is used. Scripts must use Intelligent CCA rather than Full CCA for the first half of a call where a secondary dialtone is required before dialing the second half of the call. Note that the secondary dialtone can only be reported when using the AYC21 circuit card (E1/T1).

5. On a E1 or T1 channel, any dial string with a character other than 1,2,3,4,5,6,7,8,9,0,#,*,A,B,C, D,a,b,c, or d is illegal. PRI channels allow all of the above characters except * and #. On Tip/Ring channels, any string with a character other than 1,2,3,4,5,6,7,8,9,0,#,*,A,B,C,D, a,b,c,d,(,), or – is illegal for touch-tone dialing. For dial pulse dialing on a Tip/Ring channel, any string with a character other than 1,2,3,4,5,6,7,8,9,0,(,), or – is illegal. For E1, T1, and PRI, the maximum dial string is 15 digits. For Tip/Ring channels, the maximum dial string is 30 digits. If more digits need to be dialed than are allowed, multiple **tic('o')**, **tic('O')**, **tic('d')**, and/or **tic('D')** instructions may be required.
6. To get the caller disconnected value ('h' for *r.0*), the Tip/Ring, LSE1, or LST1 channel must be connected to a Lucent Technologies DEFINITY ECS or compatible switch with the optional feature Sending DTMF Feedback Tones to the VRU properly administered on the switch and on the VRU.

You should be aware of the following issues when using these dispositions:

- Fast busy ('F') represents any temporary error condition not explicitly listed; for example, congestion or circuit busy, as well as fast busy. A blind success (0) means that the call was dialed successfully but that the system does not know if the call was answered.
- Stutter dialtone ('S') is generated by some switches in response to a flash. When this disposition is returned to the script, it means that dialing can proceed. For these switches, any other call disposition when the switch is flashing indicates an error.
- A timeout (-2) means any type of timeout including a timeout on a **tic** instruction or on the classifier (the SSP or SP).

- When v, p, D, or F is returned for outdials on PRI channels, more information (the ISDN cause values) on the call dispositions for PRI channels is available in register 1 (*r.1*). Table on page 560 lists the ISDN cause values.
- When Register *r.0* is set to ‘s’, the Special Information Tone (SIT) is in *r.1*.

Example

The following example (Figure 32) is an excerpt from a script showing how a developer might use the setcca and **tic** instructions in a Full CCA application.

Figure 32. PRI Application Using the tic Instruction

```
setcca(1,10,-1)
nextcall:
dbase( .... )     /* get number to dial from DIP */
tic(‘O’, r.3)    /* call number in register 3 */
jmp(r.0 == ’N’, noAns)     /* no answer after 10 rings */
jmp(r.0 == ’B’, busy)
jmp(r.0 == ’F’, retry)
jmp(r.0 == ’A’, answer)
jmp(r.0 == ’s’, SIT)
jmp(r.0 == -4, noResource)
noAns:
tic(‘h’)     /* put line on-hook to stop ringing */
busy:
dbase ( .... )    /* report result to controlling DIP */
goto (nextcall)
```

```
SIT:
jmp(r.1 == 'R', retry)
jmp(r.1 == 'r', retry)
jmp(r.1 == 'K', retry)
jmp(r.1 == 'k', retry)
dbase ( .... )     /* report result to controlling DIP */

answer:
talk("Hello, you may be the winner of a free trip to Hawaii")
dbase ( .... )     /* report result to controlling DIP */
goto (nextcall)
```

tnum

Name

This script instruction speaks a number with natural speech.

Synopsis

tnum(***type.src{[,'inflection'],[type.style]}***)

Description

The **tnum** instruction accepts the numeric value specified by the first argument, translates it into a string of phrases, and puts it in a queue for speaking. The second argument, when specified, controls the speech inflection. The optional *type.style* argument specifies the coding style.

The **tnum** instruction does not support speaking numbers in the billions and trillions because most of these numbers are too big to fit into an integer variable. However, the phrases "billion" and "trillion" are included in the Enhanced Basic Speech package. If your script requires such large numbers,

we suggest that you start with an ASCII string, parse the string (getting the amounts of billions and trillions as substrings), then convert the three resulting substrings to integer values and speak them with the tnum instruction. Insert a **talk** instruction with the phrase for “trillion” or “billion,” where appropriate.

Example

In the following example, the program says it could not understand the caller. Then the system pauses for 500 milliseconds of silence, and asks if the number is the n-digit number that is stored in *r.1* or the number “six-hundred-fourteen.” The **tnum** instruction says the number to the caller in natural-sounding speech.

The **atoi** conversion instruction is needed since the **getinput** instruction returns information as a null terminated character string, but the **tnum** instruction uses integers.

The **tnum** instruction is used when it is desired to hear the words hundred, thousand, thirty, etc. in the response. The **tchars** instruction differs from **tnum** by only speaking the numbers individually.

```
GET_ID:
 getinput(ch.DG,9)
 atoi(r.1,ch.DG)
   ..............  ..............................
 talk("i could not understand you","sil.500","did you say")
 tnum(r.1)
 talk("or")
 tnum(614)
   ..............   ..............................
```

trace

Name

This script instruction works with the trace line instruction to monitor scripts.

Synopsis

trace*(type.src*[*,type.src])*

Description

The **trace** script instruction works with the trace line command to display a message from the script if the trace command is run on the channel on which the script is running. These trace messages allow application developers to monitor the progress of a script. This capability is useful in debugging and troubleshooting scripts, either during the initial application development or if problems rise while the application is running. The **trace** instruction allows TSM to print messages to the shared memory area for trace messages. These messages can include the default trace messages for TSM or for a

specific channel. The **trace** command is discussed in *UCS 1000 R4.2 Administration*, 585-313-507.

Note: If there are too many traces running simultaneously on a system, the buffer in which this information is stored may be filled and some data lost, with no notice of this in the trace output.

The first argument is evaluated as a number and is used as a step identifier. The optional argument can be used to print a specific data value of interest. The optional argument may be any integer type or a null terminated character string.

Examples

The instruction

trace(1000)

in a script running on channel 2 produces the following line in the output of the trace command if it is being run for that channel.

`CH002: <script>: STEP: 1000.`

where *<script>* is the name of the script running on channel 2.

The instruction

trace(1000, "Accessing Customer Database")

in a script running on channel 21 produces the following line in the output of the trace command if it is being run for that channel.

CH021: <script> STEP: 1000 VALUE: Accessing Customer Database

See Also

talk
tchar
tflush

tstop

Name

This script instruction stops play on a conversation.

Synopsis

tstop([*wait*])

Description

The **tstop** instruction lets the script programmer stop any voice function (playing, coding, text-to-speech) on the script's current play conversation.

The optional *wait* argument can be used to cause the script to wait for voice activity to cease before continuing. Valid values for the *wait* flag are 0 (the default) to not wait and 1 to wait. If the *wait* flag is set, the successful return values of the stop instruction depend on what voice function has been stopped. If the wait flag is not set, the tstop instruction returns 0 in *r.0* for success or -1 for failure.

If voice coding has been stopped, script *r.0* contains the phrase number (a positive integer) of the coded phrase, *r.1* contains the phrase length, and *r.2* is set to 1 (see the vc instruction). Otherwise, *r.0* will be set to 0 and *r.1* and *r.2* will not be changed.

It is strongly recommended that the **tstop** instruction *always* be used with the wait flag set to 1. This ensures that any voice activity on the channel is stopped before the script continues execution. Failure to wait may leave TSM in a state that causes subsequent operations of the script (playing, coding, dialing, etc.) to fail. The default (no wait) is supported for older scripts which depend on its behavior.

One example where it may be useful to use **tstop** without the wait flag set is in a script event interrupt routine (see the **event** instruction). If this is done, the interrupt routine should not perform any instructions involving voice activity or telephony functions after doing the **tstop** (no wait). It also should return with *r.0* set to a non-negative value so that original wait continues at the point of interruption (see the **rts** instruction). The script will continue from this point once the speech activity has actually stopped.

Example

In the following example, the script plays the phrase “music” while it processes the transaction with the host computer. After this processing completes, the music is stopped, and the caller is informed of the results.

```
   ..................   ...................

talk(int.MUSIC) /* Play music to the caller */
tflush(1,1) /* Do not let touch tones turn off music and don’t
```

```
wait */
dbase(0,FUDB,ch.ACCOUNT_ID,8,int.SELL_PRICE,4) /* Get info from
host */

tstop(1)
talk("Your account has now been credited with Lucent
Technologies stock for the price of")
tnum(int.SELL_PRICE)

   ......................   ...................
```

See Also

tflush
vc()

ttclear

Name

This script instruction clears the touch-tone buffer.

Synopsis

ttclear()

Description

The **ttclear** instruction clears the touch-tone buffer. This instruction is useful for applications in which you want to throw away all “typed ahead” input. Ttclear removes any touch tones in the touch-tone buffer when the instruction is executed. The number of touch tones cleared is stored in Register 0.

Example

```
ttclear()
```

ttdelim

Name

This script instruction defines touch-tone key functions.

Synopsis

ttdelim(*erase-char,erase-all,delim1,delim2*)

Description

The **ttdelim** instruction sets four control functions and the touch tone keys used by the caller to perform those functions. The functions for the *erase-char* and *erase-all* arguments are defined by the system; the functions for the *delim1* and *delim2* arguments are defined by the developer. The touch tone keys for performing all four functions are defined by the developer.

The system-defined functions *erase-char* and *erase-all* do not terminate the collection of touch tones initiated by the **getinput** or **getdig** instruction and those characters are removed from the buffer; whereas, the developer-defined functions *delim1* and *delim2* terminate the collection of touch tones and those characters remain in the buffer.

The touch-tone buffer is scanned for the delimiters currently in effect when a **getinput** or **getdig** instruction is executed rather than while the touch tones are entered.

The values for the ttdelim arguments are:

Value	Meaning
-1	Function is not used (default)
0	Do not change value of current function
'c' or 'cc'	New value where c is: 0–9, #, *, or A–D (only on extended keypad, such as an operator console)

The following functions and characters might be specified for the instruction:

```
ttdelim('#1','#*','*1','*2')
```

Characters	Meaning
#1	Erase one character
#*	Erase all characters
*1	Get operator
*2	Play help message

If a script does not use the **ttdelim** feature, then this instruction is not used. On the other hand, if it uses only one argument, then a default value must be entered for the other three arguments.

An example of a ttdelim instruction using only the *erase-all* function is:

```
ttdelim(-1,'#',-1,-1)
```

To allow for the extra digits requested by a *delim1* or *delim2* argument, the **getinput** or **getdig** instruction should specify more digits than it needs. For instance, if a 5-digit entry is required, but it is anticipated that a caller might enter all incorrectly, and need to erase them, **getinput** or **getdig** would require a minimum of seven digits.

The **ttdelim** instruction works with the **getinput** (or **getdig**) and **tttime** instructions. For example, after requesting 5 digits with a **getinput** or **getdig** instruction, normally *r.0* is set to 5 and the actual digits received are stored at the destination. Whenever the **ttdelim** instruction is used, the **getinput** or **getdig** instruction has to check the values of *r.0* and the received digits to determine if *delim1* or *delim2* was used.

Based on the previous arguments for the ttdelim instruction, the **getinput** or **getdig** instruction would have the results given by the following examples.

Input	*r.0*	Destination	Script Action
12345	5	12345	Use 5 digits
123#*45678	5	45678	Use 5 digits
12*1	4	12*1	Transfer to operator
*1	2	*1	Transfer to operator
12*2	4	12*2	Play help message and reprompt for input

The timeouts for the two system defined functions, *erase-char* and *erase-all*, are the same. The **tttime** instruction only uses the firstdig argument once, but it repeatedly uses the *interdig* argument to wait the maximum amount of time specified to receive the next digit.

The script writer needs to write code to implement the functions. For example, *delim2* would need a **talk** instruction to play the help message.

Example

The following example causes the last digit to be erased when the #1-keys are pressed, the entire entry erased when the “#” key is repeated, and the entry terminated when the *-key is pressed. The value -1 indicates that the fourth argument is not used.

```
ttdelim('#1','##','*',-1)
```

Note: Difficulty could arise with conflicting definitions (for example, # and #1). Since both a single character string and double character string are permitted, the system may respond when the first # key is pressed and never read the second key.

Note: Any additional significance attached to the "*" key entry, other than entry termination, must be written into the script.

Note: The digits entered at the delimiter must be accounted for in the **getinput** or **getdig** instruction. For example, if 8 digits plus a touch-tone terminator are expected, **getinput** or **getdig** must look for 8 digits plus the length of the touch-tone terminator.

ttintr

Name This script instruction selectively identifies touch-tones that interrupt speech

Synopsis **ttintr(ctype.ttstr, ctype.prev_ttstr)**

Description **ttintr()** sets the TouchTone (TT) interrupt mask to the string indicated by *ctype.ttstr* and copies the previous mask value to *ctype.prev_ttstr*. The TT interrupt mask determines what touch-tones interrupt speech playing or coding for the script. If char.-1 is used for *ctype.prev_ttstr*, then the previous mask value is not returned. If char.-1 is used for *ctype.ttstr*, then the current mask value is not changed.

Return Values **ttintr()** returns 0 in register 0 (*r.0*) for success and -1 for failure.

Examples The following example sets interrupt mask so only digits 1, 2 or 3 interrupt coding or playing, and stores previous mask at the address defined by PREV_INTRMASK.

```
ttintr("123", char.PREV_INTRMASK)
```

See Also **ttmask()**

ttmask

Name This script instruction selectively identifies touch-tones that can be reported

Synopsis **ttmask(ctype.ttstr, ctype.prev_ttstr)**

Description **ttmask()** sets the TouchTone (TT) input mask to the string indicated by *ctype.ttstr* and copies the previous mask value to *ctype.prev_ttstr*. The TT interrupt mask determines what touch-tones are received as input by the script. If char.-1 is used for *ctype.prev_ttstr*, then the previous mask value is not returned. If char.-1 is used for *ctype.ttstr*, then the current mask value is not changed.

Return Values **ttmask()** returns 0 in register 0 (*r.0*) for success and -1 for failure.

Examples

The following example sets input mask so only digits 2, 4, 6, or 8 are received, stores previous mask at address defined by PREV_INPMASK.

```
ttmask("2468", char.PREV_INPMASK)
```

The following example sets the mask back to its previously stored value.

```
ttmask(char.PREV_INPMASK, char.-1)
```

See Also

ttintr()

tttime

Name

This script instruction sets the time-out values for touch-tone input.

Synopsis

tttime(*type.firstdig,type.interdig*)

Description

The **tttime** instruction allows a script to set the touch-tone timeout values. *Firstdig* specifies the maximum seconds that the system should wait to receive the first touch-tone digit after executing a **getinput** or **getdig** instruction. *Interdig* specifies the maximum seconds to wait between two consecutive touch tone inputs. The default values are 10 seconds to wait for the first touch-tone digit and 10 seconds to wait between consecutive touch tones. There are no limits to timeout times.

The **tttime** instruction is related to the **getinput** or **getdig** instruction. If the *firstdig* time is exceeded, *r.0* is set to 0 and the **getinput** or **getdig** instruction terminates. If the *interdig* time is exceeded, *r.0* is set to the number of digits that are received, transferred to the script buffer, the **getinput** or **getdig** instruction terminates.

Example

In the following example, before asking a caller for a response, the **tttime** instruction sets the system to wait no more than 4 seconds for the caller's initial response and up to 2 seconds between digits before automatically returning from the data gathering instruction.

```
GET_IO_MODE:

 tttime(4,2)
 talk("enter your id now")
 ..................................  ............. ........
```

vc

Name

This script instruction codes a phrase and stores it in a talkfile.

Synopsis

vc(*flag,type.time,type.rate*[,*wait_flag*])

Description

The **vc** instruction codes speech into a phrase in a talkfile. For the *flag.type* argument, 'b' (for "begin coding") is accepted. Another character value, ''p' (for prompt) may be used with the SSP circuit card to play a short beep just before voice coding starts. Note that this beep must be included in the prompt, either as a separate phrase or as part of the phrase.

Note: Gain control does not work for the LSPS II circuit card using this instruction.

The *type.time* argument specifies the maximum duration, in seconds, of the coding session. A value 'n' for *type.time* specifies a coding session lasting up to 'n' seconds. A value of -1 or 0 for type.time specifies the default maximum duration of 45 seconds. Coding can be terminated at any time by entering a touch tone.

The *type.rate* argument specifies the coding rate in kilobits per second. The valid coding types and rates are defined in the header file *codestyle.h*. If the value given for this argument is not a valid rate or type, the instruction fails. In addition to coding types, two modifiers are defined in **codestyle.h** that can be used to turn off silence trimming (NO_SIL) and automatic gain control (NO_AGC) during voice coding. To use these features, the corresponding modifier must be ORed together with the coding type. The following examples show how to start voice coding for a maximum of 60 seconds using the ADPCM32 coding type and no automatic gain control:

```
load(int.CODETYPE, ADPCM32)
or(int.CODETYPE, NO_AGC)
vc('b', 60, int.CODETYPE)
```

The default value for the optional *wait_flag* argument is 1, which causes **vc()** to return when voice coding is complete. If this argument is used with a value of 0, **vc()** will return immediately after voice coding has started, allowing the script to execute more instructions while doing voice coding. This is useful for doing voice coding and speech recognition (with the **getinput()** instruction) simultaneously. The script can record what the caller is saying while it is being recognized.

When voice coding is started without waiting for completion, **vc()** returns a value of 0 in register 0 (*r.0*). Voice coding must be stopped at a later time by the **tstop()** instruction with the optional *wait_flag* argument set to1 to get the return values (*r.0, r,1*, and *r.2*) from the completed voice coding.

If the **vc** instruction is successfully completed, it returns the phrase id in register 0. If the vc instruction is not successfully completed, it returns a negative value in register 0. A -1 in register 0 means coding failed, -2 means the initial silence timeout set by **vctime** was exceeded. When voice coding is started without waiting for completion, **vc()** returns a value of 0 in register 0. Register 1 contains the recorded message length in seconds. Register 2 is set to 1 if coding completed normally, 2 if was coding terminated by touch tone, and 3 if the intermediate silence timeout set by **vctime** is exceeded.

Examples

In the following example, a beep will sound, then a phrase will be coded for a maximum of 100 seconds using ADPCM at a rate of 32Kbps.

```
load (int.TIME,100)
vc (’p’,int.TIME,ADPCM32)
```

In the following example, a phrase will be coded for a maximum of 120 seconds using sub-band coding at a rate of 16Kbps. No beep will sound.

```
load(short.RATE,SBC16)
vc('b',120,short.RATE)
```

In the following example, the **vc()** instruction uses the optional *wait_flag* argument to allow voice coding and recognition at the same time.

```
#include "irDefines.h"
#include "/att/include/codestyle.h"
/* Answer phone */
tic('a')
/* Reserve a phrase */
phreserve(65535, 103, 20, ADPCM32)
jmp(r.0 < 0, L__quit)
load(int.PHRASE, r.1)/* phrase id */
load(int.TFILE, r.0)/* talk file number */
/* Play prompt.  Must play prompt before voice coding
 * is started.  Coding and playing cannot be done
 * simultaneously.  This means that barge-in can't be
 * done while coding and recognizing simultaneously.
 */
talk("5")
tflush(1)
```

```
/* no silence timeouts (or, silence timeouts must be
 * significantly greater than getinput timeouts (see
 * tttime instruction)
 */
vctime(0, 0)
/* set input timouts */
tttime(5, 5)
/* start coding with no wait.
 */
vc('b', 20, ADPCM32, 0)
/* get recognition input form caller. */
recog_start(IRD_WHOLE_WORD, US_4dig, 0)
getinput(ch.CI_VALUE, 4)
load(int.NUM_DIGS_GOT, r.0)
/* stop voice coding  and wait for code complete*/
tstop(1)
/* return vales from tstop() are the same as for vc().
 */
load(int.PHRASE, r.0 )
load(int.LENGTH, r.1 )
load(int.STATUS, r.2 )
L__quit:
quit()
```

See Also **tstop()**

vctime

Name

This script instruction sets the silence timeouts for voice coding.

Synopsis

vctime([*type.src*] [*type.src*])

Description

The **vctime** instruction allows the script writer to set silence timeouts. The first *type.src* argument contains the value for the initial silence timeout. The second *type.src* argument contains the value for the inter-word silence timeout. The maximum timeout is 30 seconds.

The values for the *type.src* arguments and the effect on the timeout are given below:

Value	Effective Timeout Value
X > 0	X becomes the timeout value
X = 0	Timeout is turned off
X < 0	Timeout is set to default value (5 seconds)

A comma or place holder with a value of 0 is used when an argument is not inserted. This instruction does not give a return value to indicate success or failure.

Example

In the following example, initial silence timeout and inter-word silence timeout are set to three seconds.

```
vctime(3,3)
```

C C-Library Functions

Overview

This appendix contains summaries of the C-library functions discussed earlier in this book. This chapter is divided into three sections according to the library in which the functions reside: libspp.so (was libspp.a), libalerter.a, or liblog.a.

The functions are listed in alphabetical order under the library in which they are located. Each function appears on a separate page including the following information:

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

Use these library pages to locate detailed information about each function.

Purpose

The purpose of this appendix is to provide application developers with the information required to use the available C-library functions to develop an application.

C-Library Function Locations

Table 35 on page 595 lists the functions in alphabetical order and the library in which the function resides. Equivalent INTUITY™ Response Application Programming Interface (IRAPI) instructions are also listed, if available.

Note: Most libspp.so functions have been replaced by IRAPI functions. All new data interface processes (DIPs) should be written in terms of the IRAPI. See Chapter 5, IRAPI, for additional information.

Note: Information about each IRAPI instruction is included in the manual pages available online on the system in directory **/vs/man/cat3** or **/vs/man/cat4**.

Table 35. C-Library Function Locations

Function	IRAPI Equivalent	Library
db_init	irRegister	libspp.so
db_pr	irTrace	libspp.so
db_put	irTrace	libspp.so
expandLog		liblog.a
logDstPri		liblog.a
logMsg		liblog.a
mesgrcv	irWCheck	libspp.so
mesgsnd	irPostEvent	libspp.so
startup	irRegister	libspp.so
threshold		libalerter.a
VSerror		libspp.so

1 of 2

Table 35. C-Library Function Locations

Function	IRAPI Equivalent	Library
VSstartup	irRegister	libspp.so
VStoname		libspp.so
VStoQkey	irGetQkey	libspp.so

2 of 2

libspp.so Functions

db_init

Name

db_init — Initialize and activate trace facility

Synopsis

```
#include <spp.h>
int db_init (qkey)
int     qkey; /* message queue key of process */
```

Description

Data interface processes (DIPs) call db_init once at the start to set up the trace mechanism provided by the system. The *qkey* is the message queue key assigned to the process.

From then on, processes can use db_pr and db_put to write out trace/debug messages. These trace messages then can be displayed on the command line by using the **trace** command (see *UCS 1000 R4.2 Administration*, 585-313-507). On behalf of the process, VSstartup and startup call db_init so this function does not have to be called directly.

Diagnostics

No indication of success or failure is returned. If it fails, the dp_put message appears on standard error (stderr) and db_pr messages are ignored.

See Also

db_pr
db_put
startup
trace
VSstartup

db_pr

Name

db_pr — Conditionally output trace message

Synopsis

```
#include <spp.h>
int db_pr (format)
char *format; /* printf(3S) format string */
```

Description

db_pr writes out the string formed using the same printf conventions to the trace buffer if the calling process now is being traced. (See the *UnixWare System Programmer's Reference Manual* for additional information on printf). If the process is not being traced, db_pr does nothing. If also does nothing if the trace facility was not initialized through db_init. db_pr calls db_put after forming the string to write out.

The db_pr messages can be displayed on the standard out (stdout) through the trace command.

Examples

db_pr("%s: Got Message on channel=%d\n", "Dip", 5);

Diagnostics

No indication of success or failure is returned.

Warning

db_pr can apply up to a maximum of nine arguments to the specified format string.

See Also

db_init
db_put
trace

db_put

Name

db_put — Unconditionally output a string to the trace buffer

Synopsis

#include <spp.h>

int db_put (string)
char *string; /* string to write out */

Description

The **dp_put** function writes the string to the trace buffer regardless of whether the calling process now is being traced. It writes the string as it is to standard error if the trace facility was not initialized through db_init. Before writing to the trace buffer, it splits the output string into 78 character lines (if necessary) to fit in the buffer.

The db_put message can be displayed on standard out (stdout) through the trace command.

Examples

db_put ("DIP: Got a Message ");

Diagnostics

No indication of success or failure is returned.

See Also

db_init
trace

mesgrcv

Name

mesgrcv — Get an IPC message

Synopsis

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include "mesg.h"
#include "spp.h"

int mesgrcv (morig,msgp,msgsz,msgtyp,msgflag,msgrtime)
int       morig
union msgunion *msgp; /* message buffer */
int       msgtyp; /* type of message to read */
int       msgsz; /* size of message buffer */
int       msgflag; /* control flag */
long      *msgrtime; /*message receive time */
```

Description

mesgrcv is used by the voice system to read IPC messages off their RM or UNIX message queues. mesgrcv reads the next IPC message on the message queue (morig) and stores up to msgsz bytes in the buffer pointed to by msgp. The buffer should be as large as the largest message to be read. Otherwise, by default, mesgrcv will discard the message if its size is greater than *msgsz*. However, if (MSG_NOERROR and msgflag) is true, the

message will truncated to fit in the buffer (see **msgop(2)** in the *UnixWare Command Reference*).

mesgrcv can read messages of a certain type selectively as specified by *msgtyp*. See *msgop(2)* for the possible values for *msgtyp*. Set *msgtyp* to zero to read the first message on the queue regardless of type.

By default mesgrcv waits indefinitely for a message to arrive on the queue if there is none currently to be read. However, if (msgflag & IPC_NOWAIT) is true, mesgrcv returns immediately with a -1 and errno is set to ENOMSG when no message is found on the queue. mesgrcv also returns (in msgrtime) the time the message was read and stored in the buffer (**msgp*) if IPC_GTIME and *msgflag* is true. The **mesgrcv** function creates the IPC message queue for queue key *morig* if necessary.

The /vs/data/unix_ipc_qkeys file can be used by add-on packages to enter the qkey number for qkeys that should have messages sent via UNIX IPC rather than RM IPC (for backward compatibility with dips that use the UNIX msgsnd and msgrcv IPC functions rather than the mesgsnd and mesgrcv functions). This backward compatibility support for use of UNIX IPC may be eliminated in future releases. Messages sent to the IRAPI channel owner must use RM IPC. Each qkey number should be on a separate line qkey numbers should be removed when the package is removed. Only numbers are supported; define symbol names are not supported.

Example

The following are examples of code fragments using **mesgrcv** to receive IPC messages. The examples assume that a TSM script is sending two types of messages. One contains the caller's personal information and the second contains the caller's order for a specified number of widgets.

```
/* Definition of the message structures and definitions
* used in the examples below.
*/
#include <sys/errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include "spp.h"
#include "mesg.h"


typedef enum {VISA,AMEX} CREDIT_CARD_TYPE;


/* Define messages to receive */
struct callerinfoMsg {
    struct mbhdr hd; /* see mesg.h */
    CREDIT_CARD_TYPE creditCard; /*VISA, AMEX */
    int creditCardNo; /* employee number */
};
#define CALLER_INFO 6910 /* message id */
```

```
struct orderMsg {
    struct mbhdr hd; /* see mesg.h */
    int noWidgetsOrdered; /* employee number */
};

#define ORDER_AMOUNT 6930 /* message id */

/* Actual area for receiving messages.
* Compose of all expected messages.
*/
union msgBuffer {
        struct orderMsg order;
        struct callerinfoMsg caller;
} MsgRcvArea;

union msgBuffer *Msgp = &MsgRcvArea;

extern int errno;

char *Myname = "WidgetDip";
int myQkey; /* my very own message queue */
int     noBytesRead;
long  Msgrtime;

/* Dummy function that contains the examples.
```

```
void
receiveExamples()
{
/*******Example I
* To read the first message on the queue and find out
* what message you got:
*/
int howMany;
CREDIT_CARD_TYPE cardType;
int cardNo;

noBytesRead = mesgrcv(MyQkey,Msgp,sizeof(union msgBuffer),
        0,0,NULL);
if (noBytesRead > 0) { /* no error */
/* Time to unpackage the message and find out
* what message arrived.
*/
switch (Msgp->order.hd.mcont) {
        case ORDER_AMOUNT:
        howMany=Msgp->order.noWidgetsOrdered;
        /* Process order */
        break;
case CALLER_INFO:
        cardType=Msgp->caller.creditCard;
        cardNo=Msgp->caller.creditCardNo;
```

```
        break;
default:
/* Unknown message received.
* Notify someone and probably go back
* and read something else.
*/
;/* null statement to make code compile for this example */
}
} else {
/* Could not read message for some reason.
* Depending on the error, might want to re-try reading.
* Check errno if noBytesRead==-1.
*/
}

/*******Example II
* To read message without waiting if there's none now.
* And to get the time the message was read:
*/
noBytesRead= mesgrcv(MyQkey,Msgp,sizeof(union msgBuffer),0,
        IPC_NOWAIT, &Msgrtime);
if (noBytesRead == -1 && errno == ENOMSG) {
        /* No message is on the message queue.
        * Do some other work and then come back and re-read queue
        */
```

```
} else if (nobytesRead > 0) {
        db_pr("%s: got message on %ld\n, MyName, Msgrtime);
        /* process message */
} else {
        /* some other error occurred */
        ; /* null statement to make code compile */
        }
}
```

Diagnostics

Upon successful completion, mesgrcv returns the number of bytes read, ranging from 1 to *msgsz*. Otherwise, one of the following negative values is returned:

- -1 — An error occurred in *msgrcv* and errno is set accordingly
- -2 — Can not create or get the message queue
- -3 — Destination *qkey* is not in the voice system range (1–95)
- -4 — Message was too big for the buffer and so it got discarded and (MSG_ERROR & msgflag) was false. No message was read. The size of the message buffer is too small.

See Also

msgop(2) (*UnixWare Command Reference*)
msgsnd(2(2)) (*UnixWare Command Reference*)
/vs/data/unix_ipc_qkeys

mesgsnd

Name

mesgsnd — Send an IPC message

Synopsis

```
#include "spp.h"

int mesgsnd (mdest, msgp, msgsz, msgflag)
int     mdest;        /* Message Qkey to send to */
union msgunion *msgp; /* message to send */
int     msgsz;        /* size of message */
int    msgflag;       /* flag for controlling send */
```

Description

mesgsnd sends the IPC message pointed by msgp to qkey mdest. The number of bytes in the message is specified by *msgsz*. The message consists of the header and data parts. The header is defined as struct mbhdr in *mesg.h*. The size of message (*msgsz*) should include the header and data parts. Mesgsnd actually sends the message by calling the UnixWare system call msgsnd. *Msgflag* is sent directly to the UnixWare system call *msgsnd*.

mesgsnd creates the IPC message queue for qkey mdest if necessary. When returning dbase messages to TSM, it is necessary to include the channel number value which originally came from TSM.

Example

The following is an example of a function that sends employee information to the TSM script running on the specified channel.

```
#include "spp.h"
#include "mesg.h"
/* Define message to send */
struct employeeMsg {
    struct mbhdr      hd; /* see mesg.h */
    char              ename[25];/* employee name */
    int               payrollNo; /* employee number */
};
#define   EMPLOYEE_INFO#7890/* message id */

extern int MyQkey;   /* sender's Qkey */

int
sendEmsg(chan, ename, enumber)
int chan;
char *ename;
int enumber;
{
Struct employeeMsg emsg;
int  retcode;

/* Package Message */
emsg.hd.mchan = chan;
```

```
emsg.hd.mtype = 1;/* should be positive non-zero */
emsg.hd.morig = MyQkey;
emsg.hd.mcont = EMPLOYEE_INFO;
emsg.hd.mseqno = 0;/* set to zero for safety */
strcpy(emsg.ename, ename);
emsg.payrollNo = enumber;

retcode = mesgsnd(TSM, &emsg, sizeof(emsg), 0);
return(retcode);
}
```

Diagnostics

Upon successful completion, *mesgsnd* returns zero (the value returned by msgsnd system call). Otherwise, one of the following negative values is returned:

- -1 — An error occurred in *msgsnd* and errno is set accordingly
- -2 — Can not create or get the message queue
- -3 — Destination qkey (*mdest*) is not in the voice system range (1–95)
- -4 — The size of the message is too small (less than 4 bytes)

See Also

msgop(2) (*UnixWare Command Reference*)

startup

Name

startup — Called once to initialize hardcoded processes to the voice system

Synopsis

#include spp.h

int startup (qkey, slot_offset)
int qkey; /* message qkey of calling process */
int slot_offset; /* used to get the slot for posting */

Description

startup registers and initializes the calling process to the voice system. It should be called once at the onset and is used by hardcoded processes (that is, those that know beforehand what message queue they will use to receive IPC messages). startup posts the process in a certain predefined slot in the Bulletin Board (BB). The calling process has some control over what slot is selected. The slot selected depends on the qkey and slot_offset specified.

Note: It is recommended that user DIPs use VSstartup when possible. See VSstartup on page 615.

It is important that processes choose a qkey and slot_offset that translates to a unique message queue and slot. The bbs command can be used to find out what processes are currently posted, to avoid interfering with another process. For DIPSs (identified by qkeys in the range from 20–54), startup selects a slot using the following equation:

slot = 32 + *slot_offset*

The *slot_offset* for DIPs must be between 0 and 34 or the *slot_offset* is set to zero. DIPs can read from the same message queue by specifying the same qkey but different slot_offsets.

In addition, startup initializes the trace facility so that the process can write out trace message using the **db_pr** family of functions.

startup is the old way of initializing hardcoded processes to the system and still is supported. New processes, however, should use **VSstartup** for initialization.

Specifically, startup does the following:

1. Calls the **db_init** library function to set up the trace facility for the calling process
2. Attaches the BB and initializes the global BB variables used by the rest of the interface functions
3. Posts the calling process in the BB base on its *qkey* and *slot_offset*

4 Acquires the process semaphore associated with the slot

5 Sets up the calling process to catch the SIGTERM and invoke the standard exit library function.

Examples

The following code fragment initializes a DIP with qkey DIP15 (as defined in **mesg.h**) and posts it in slot 47.

```
#include mesg.h
#include spp.h
/* No need to check the return code from startup since
* it is successful if it returns at all.
* DIP15-DIP0 is actually the DIP number for the DIP (15).
* Remember slot = 32 + DIP15–DIP0 = 32 + 15 = 47.
*/
(void) startup (DIP15, DIP15-DIP0);
```

Diagnostics

Upon successful completion, startup returns zero. It does not return if unsuccessful. startup writes out a trace message and terminates the calling process if an error is encountered. Possible errors include:

- Can not attach to the BB
- Can not create the process semaphore

However, in one case startup does not terminates. If another process is already posted in the slot, startup waits indefinitely until the process can post itself in the slot that already contains a posted process.

See Also

db_pr
trace
VSstartup

VSerror

Name

VSerror — Get text for voice system error messages

Synopsis

```
#include <sys/types.h>
#include VS.h
char *VSerror (errid)
int  errid:        /* negative error value */
```

Description

VSerror returns a character string explaining what the specified error id means. Error ids equal to negative one (-1) are treated as UnixWare system errors and the appropriate text in the UnixWare error table (sys_errlist[]) is returned. Error ids defined in VS.h and UnixWare system errors have text associated with them; all other errors are unknown to VSerror and result in a generic UNKNOWN message being returned. Currently, the error return values from VSstartup and VStoqkey and from the underlying Bulletin Board interface functions are recognized by VSerror.

Examples

```
char     *emsg;
key_t Qkey;
/* find the qkey of the speech recognition dip */
Qkey = VStoqkey(spRecog);
if (Qkey <= 0) {
        emsg = VSerror(Qkey);
        fprintf(stderr,VStoqkey failed; %s\n, emsg);
}
```

See Also

intro

VSstartup

Name

VSstartup — Called once to initialize process to the voice system

Synopsis

```
#include <sys/types.h>
#include VS.h

key_t VSstartup (procName, instance, flag)
char *procName;        /* name associated with process */
short instance;        /* process instance */
long flag;          /* Is process a DIP? */
```

Description

VSstartup is called once to initialize a process to the system. VSstartup returns the DIP name, its instance, and a DIP flag. The DIP flag can take one of two values, constants DIP_PROC or NONDIP_PROC. Setting the flag to the constant DIP_PROC allows the DIP to send messages to and receive messages from TSM scripts. If the flag is set to the constant NONDIP_RPOC, messages sent by the IDP to TSM scripts are ignored by TSM.

Processes specifying the same *procName* and difference instance numbers will be assigned the same message queue key to read from, but will be posted in separate Bulletin Board (BB) slots.

The *instance* can be any arbitrary value in the range from 0 to 32767. However, the *instance* should be unique across processes using the same

procName. A common use of the *instance* number is to differentiate between multiple copies of a process.

Specifically, VSstartup does the following:

1. Attach the BB and initialize the global BB variables used by the rest of the interface functions
2. Post the calling process in the BB and get its dynamically assigned Qkey
3. Acquire the process semaphore associated with the slot
4. Calls the db_init library functions to set up the trace facility for the calling process

Upon encountering an error, VSstartup will immediately return a predefined negative value.

Example

```
/* Post instance 0 of process xferdip as a DIP */
#define TRANSFER_DIP                “xferdip”
key_t Qkey;
char *emsg;
Qkey = VSstartup(TRANSFER_DIP, 0, DIP_PROC);
if (myQkey <= 0) {
     db_pr("%s: Can’t get qkey: VSstartup: %s\n”;
          VSerror(myQkey));
     logMsg(APPL_INITFAIL,EL_FL,Myname,"Can’t get qkey");
```

```
        sleep(5); /* to slow down continuous respawning */
        exit(1);
}
```

Diagnostics

Upon successful completion, the assigned Qkey is returned. Errors in the VSstartup indicate failure in assigning a message queue key, in which case one of the following negative values is returned.

VS_EINVAL*procName* argument cannot be NULL

VS_ELENLength of *procName* is out of range

VS_ERESV*procName* is reserved for hardcoded processes

VS_ENOPRTNon-printable character found in *procName*

VS_ENUM*Instance* is negative or out of range

VS_BADPROCAnother process with the same *procName* and *instance* is running already

VS_ENOFREENo BB slots available for posting process (see troubleshooting information in Chapter 4, Data Interface Processes for more information)

VS_ESHMATCan not attach the BB shared memory

See Also

VSerror

VStoname

Name

VStoname — Find the procName of the given message queue key

Format

```
#include <sys/types.h>
#include <stdio.h>
#include VS.h

char *VStoname(Qkey)
key_t Qkey;     /* message queue key */
```

Description

VStoname searches the voice system Bulletin Board (BB) and returns a pointer to the *procName* associated with the specified message queue key. VStoname will return the *procName* of hardcoded processes (processes not using dynamic Qkey numbers) as well as dynamic processes. If the message queue key is not found, VStoname will return a NULL. VStoname will also return a NULL if the BB is not attached using BBattach.

VStoname attaches the BB if not attached through VSstartup. Before returning it detaches the BB if it attached it to begin with.

Warning

The returned procName pointer refers to a static area whose content is overwritten by each call to VStoname.

See Also

VStoqkey

VStoqkey

Name

VStoqkey — Find the message queue key of the given process name

Synopsis

```
#include <sys/types.h>
#include VS.h

key_t VStoqkey(procName)
char    *procName;          /* unique name associated with process */
```

Description

VStoqkey searches the voice system Bulletin Board (BB) and returns the message queue key (Qkey) associated with the specified *procName*. If the *procName* is not found, VStoqkey will assign an unused Qkey and BB slot to the *procName*. The slot is then partially posted with the *procName* and Qkey. VStoqkey will return the Qkey of hardcoded processes (processes not using dynamically assigned Qkey numbers) as well as of dynamic processes.

VStoqkey waits to acquire the lock (process semaphore) on the BB before searching and writing, and releases the lock before returning to the calling routine.

VStoqkey attaches the BB if not attached through VSstartup. Before returning it detaches the BB if it attached it to begin with.

CAUTION:

Each call to VStoqkey involves a linear scan of the Bulletin Board. Therefore, it is recommended that a process call VStoqkey once for each procName it expects to reference and internally stored the returned Qkeys before entering its main processing loop. From then on, VStoqkey need not be invoked since the Qkeys are already available (see example for VStoqkey).

Also be aware that a process conceivably could use up all the VS message queues by repeatedly calling VStoqkey with nonexistent procNames.

Examples

```
main () {

    key_t DBDIPQKEY;
    key_t VCTDIPQKEY;
    key_t BRIDGEDIPQKEY;
    key_t myQkey;
    char *emsg;
    int nbytesRead;
    long rcvtime
    struct ms_univ msg; /* see mesg.h */

    /* Post myself in BB and init BB global variables */
    myQkey = VSstartup("CallBridge", 0, DIP_PROC);
    if (myQkey <= 0) {
        emsg = VSerror(myQkey);
```

```
        fprintf(stderr,"VSstartup failed: %s\n", emsg);
        sleep(20); /* sleep to avoid continuously respawning. */
        exit(1);
    }
    DBDIPQKEY = VStoqkey(dbdip);
    VCTDIPQKEY = VStoqkey(vctdip);
    BRIDGEDIPQKEY = VStoqkey(bridgedip);
    if (DBDIPQKEY < 0 || VCTDIPQKEY <0 ||
            BRIDGEDIPQKEY < 0) {
    /* Could not get Qkey */
    /* Report the using VSerror, et_send error and cleanup */
    sleep(10); /* to slow down continuous respawns. */
    exit(1);
    }
    /* main processing loop */
    while (1) {
        /* read next message queue and switch on sender Qkey */
        nbytesRead = mesgrcv(myQkey, &msg, sizeof(msg), 0, 0,
            &rcvtime);
    switch (msg.hd.morig) {
        case DBDIPQKEY:
        /* process message from Database DIP */
        break;
    case VCTDIPQKEY:
      /* process message from Voice Coding DIP */
```

```
        break;
      case BRIDGEDIPQKEY:
        /* process message from bridging DIP */
        break;
      default:
        /* unknown sender */
        break;
          }
      }
}
```

Diagnostics

Upon successful completion, the Qkey value is returned; otherwise one of the following errors is returned:

- VS_EINVAL *procName* argument cannot be NULL
- VS_ELEN Length of *procName* is zero or greater than 15 characters in length.
- VS_ERESV *procName* is reserved for hardcoded processes and the specified *procName* is not posted already
- VS_ENOPRT Non-printable character found in *procNam*
- VS_ENOFREE No BB slots available for posting process (see troubleshooting information in Chapter 4, Data Interface Processes for more information)

libalerter.a Function

The following information is provided in this appendix for completeness. This function is not required for proper operation of the TAS script or DIP, but some information contained here may be useful.

threshold

Name

createMsgCarrier, freeMsgCarrier, ckMsgsOfThreshold, cleanThresholds, mkThreshold, freeThreshold, threshold, findThreshold, resetThreshold, printThreshold, setThresholdCleanupInterval — Alerting message threshold management routines

Synopsis

```
#include <stdio.h>
#include <time.h>
#include “primsdefs.h”
#include “logMsg.h”
#include “threshold..h”

struct MsgCarrier *createMsgCarrier ( headp,lmp )
struct Threshold **headp ‘
struct LogMsg *lmp ;

void freeMsgCarrier ( headp,msgp )
```

```
struct Threshold **headp ;
struct LogMsg *lmp ;

void ckMsgsOfThreshold ( thp )
struct Threshold *thp ;

void cleanThresholds ()

int setThresholdCleanupInterval ( newValue )
int newValue ;

typedef void (*DownFunc)(struct Threshold *thp,int previousLevel) ;

int mkThreshold (
msgID,name,period,flags,downFunc,applPtr,cnt[level,...])
int msgID ;/* Index of msg of interest. */
int period ;/* Threshold period in seconds. */
int flags ‘
DownFunc downFunc ;/* Function called when activation level
      * decreased. */
POINTER applPtr :/* Pointer to application specific
      * information about threshold. */
int cnt ;/* # of threshold levels. */
int level ;/* Threshold level as msg count. */
```

```
voicd freeThreshold (thp ) ;
struct Threshold *thp ;

int threshold ( thp,lmp )
struct Threshold *thp ;/* Threshold description. */
struct LogMsg *lmp ; /* Parsed logging message. */

struct Threshold *findThreshold ( name )
char *name ;

int resetThreshold  (name )
char *name ;

int printThreshold (name,fp )
char *name ;
FILE *fp ;
```

Description

A Threshold structure is designed to keep track of a number of compressed logged messages with respect to a specified time period that terminates at the current time.

```
struct Threshold
        {
        struct Threshold *th_next ;
        struct Threshold *th_prev ;
```

```
        char *th_name ;/* ASCII name of this threshold */
        int th_msgID ;/* Message ID of interest. If -1, then
         * any message is acceptable. */
        int th_period ;/* # of seconds over which threshold is computed. */
        int th_flags ;

#define TH_EDGE_TRIGGERED0x0000/* Respond when level exceeded
*/
#define TH_LEVEL_TRIGGERED0x0001/* Respond whenever above
level */
#define TH_NO_REPORT0x0002   /* Suppress threshold level
          * changes messages. */

        DownFunc th_downFunc ;/* Function to call when drops in
          * levels of activation. */
        POINTER th_applPtr ;/* Pointer used by application to
         * point to information not
         * contained in the Threshold
         * structure. It is assumed that
         * this pointer will always be type
         * cast to an appropriate type. */
        int th_levelCnt ;/* Number of levels */
        int *th_levels ;/* Array of levels */
        int *th_curLvl ;/* Pointer to the current level */
        int th_msgCnt ;/* # of messages currently in storage */
```

```
        struct MsgCarrier *th_msgs ;/* Copies of messages currently
         * saved for this threshold. */
} ;
```

mkThreshold () is used to create a new Threshold structure and link it into the list of active thresholds. *msgID* is either -1, meaning that this threshold will take any message given to it, or the value of one specific message index that is to be accepted by this threshold. *name* points to an ASCII string, which is used to identify this threshold to the external world. It should be unique from all other threshold names. *period* is the number of seconds that a message will be retained by the threshold before it is discarded. *flags* is used to identify the characteristics of the threshold. In particular a threshold may be *edge* triggered, meaning it responds only when the number of stored messages crossed a boundary between activation levels, or it can be *level* triggered, meaning that whenever a message arrives and the number of messages is in excess of an activation level, there is a response. To clarify, consider a threshold that has activation levels of 3 and 6 messages. If it is edge triggered, it will only respond when the number of stored messages goes from 2 to 3 or from 5 to 6. If it is level triggered, it will respond with level 1 when messages 3, 4, and 5 arrive and with level 2 whenever the number of messages is 6 or more. By default, thresholds are edge triggered unless the TH_LEVEL_TRIGGERED flag is included in the *flags* word. In addition, it is possible to suppress reports of activation level changes by including the TH_NO_REPORT flag. Normally, whenever a threshold is crossed, a message is automatically generated reporting this fact. These messages are generated both on the way up and on the way down. When these messages

are generated, examining the log files will tell you the activation level of each threshold. When messages age enough, they are removed from storage within the Threshold structure. If this causes a threshold to drop from one activation level to a lower activation level, a message will be generated if **TH_NO_REPORT** is *NOT* specified and, if not **NULL**, the function specified by **downFunc** will be called as

*(*downFunc)(thp.previousLevel);*

This function can handle any application specific activity that is appropriate as a threshold drops from a higher activation level to a lower level, such as turning off a light or alarm indicator. If there is additional information that needs to be stored with a threshold, it can be joined to the Threshold structure via the applPtr. The form and management of this data is entirely the responsibility of the application. The pointer is stored in the Threshold structure in the *th_applPtr* element. Each threshold has one or more activation levels. *cnt* specifies the number of different thresholds. Following *cnt* will be that number of threshold levels. They are assumed to be in ascending order. Each *level* is the number of messages that must be stored in the Threshold structure for the threshold to be at that specific activation level. At least one level must be specified and it may be 0. mkThreshold () links the newly created Threshold structure into the list of active threshold specifications. If it is the first specification, a timer is started, which will continuously clean up messages as they age. Also the address of the new Threshold structure is returned so that it can be used in calls to threshold ().

A Threshold structure can be removed from the active list and its resources released via freeThreshold (). If any cleanup is required for the *th_applPtr*, it is assumed to have been performed by the caller prior to the call to freeThreshold ().

threshold () compares the index of the message described by *lmp* with the *th_msgID* element of the Threshold structure. If the *th_msgID* is -1 or if the index matches *th_msgID*, threshold () stores the message in the Threshold structure. If the addition of a new message requires an action based on the type of threshold, threshold () will return the current activation level of the threshold. If not action is required, threshold () returns 0. It is the responsibility of the calling function to respond appropriately to a non-zero response by threshold ().

lmp is a pointer to a parsed logging message, as returned by either readParsedLogMsg () or parseCmpLog (). (See readLog (3x).)

threshold () returns 0 whenever a new message is stored and the threshold still has not risen to its specified first level of activation, or for EDGE triggered thresholds, whenever the new message is not causing a level transition from a lower level of activation to a higher level of activation. Any real response to a change in activation levels of a threshold is the responsibility of the calling routine. Its response should be dictated by the activation level returned by threshold (). The only response provided automatically by threshold () is the generation of the "change of activation level" message, if not suppressed.

Any message that is retained in the threshold, either because *thp->th_msgID* was -1 or because the index of the message matched *thp->th_msgID*, is stored in an allocated *MsgCarrier* structure which has the following form:

```
struct MsgCarrier
    {
    struct MsgCarrier *mc_next ;
    struct MsgCarrier *mc_prev ;
    time_t mc_timeStamp ;/* Time stamp associated with msg */
    int mc_msgID ;/* Extracted ID of msg */
    char *mc_cmpMsgPtr ;          /* Compressed msg */
} ;
```

These *MsgCarrier* structures are linked together in a circular list, from oldest to newest in terms of order of receipt. It is assumed that the messages arrive in time stamped order.

MsgCarrier structures are created and linked into the circular list associated with a Threshold structure using the createMsgCarrier () routine. *headp* points to the current head of the circularly linked list. If it points to a NULL pointer, then the newly created *MsgCarrier* structure becomes the head of the list. freeMsgCarrier () removes a *MsgCarrier* from a circularly linked list and frees the associated storage. It is assumed that the message pointed to by *msgp* is one of the *MsgCarrier* structures associated with the list pointed to by *headp*.

ckMsgsOfThreshold () scans all the messages currently associated with the Threshold structure pointed to by *thp*, and removes and discards any that are older than the time period of the threshold. If this causes the threshold to drop one or more levels, then appropriate logging messages are generated unless the TH_NO_REPORT flag is set. ckMsgsOfThreshold () is automatically called by threshold () whenever it decides to store a new message. This insures that only messages within the current time period are on the threshold when the new message is stored.

cleanThresholds () is a timeout routine, arranged to be called by threshold () once every 60 seconds or the value specified by setThresholdCleanupInterval (). cleanThresholds () scans all Threshold structures for messages that have gotten too old and removes them, logging appropriate level drops when appropriate. Once started, it automatically reschedules itself according to the current interval value. setThresholdCleanupInterval () changes the cleanup interval to the number of seconds specified by *newValue*. The previous value is returned.

findThreshold () scans the list of active thresholds and returns the one whose name matches *name*. NULL is returned if the proper threshold cannot be found.

resetThreshold () causes all messages currently stored within a Threshold structure to be discarded. If *name* is NULL, all thresholds are reset otherwise only that threshold specified by *name* is reset. Resetting a threshold does result in an appropriate log message if the level of the threshold changes. FALSE is returned if the specified Threshold structure cannot be found.

printThreshold () causes all messages currently stored within a Threshold structure to be printed in addition to the parameters associated with the threshold. All printing is done to the standard I/O stream specified by *fp*. If *name* is NULL, all thresholds are printed otherwise only that threshold specified by *name* is printed. FALSE is returned if the specified Threshold structure cannot be found.

Caveats

Use the fast malloc () routines found in -lmalloc because they perform a lot of allocating and deallocating.

See Also

readLog (3x)
timeDesc (3x)

liblog.a Functions

The following information is provided in this appendix for completeness. These functions are not required for proper operation of the TAS script or DIP, but some information contained here may be useful.

expandLog

Name

expandLog — Expands a compressed log string

Synopsis

```
char *expandLog ( cmpmsg )
char *cmpmsg ;

char *getExpandFmt ( index )
int index ;

void endExpandFmt ()

void setExpandFmt ( file )
char *file ;

char *getExpansionFmt ()

void chgExpansionFmt ( fmt )
char *fmt ;

int getExpansionCutoff ()

int chgExpansionCutoff ( newcol )

int newcol ;
```

char *getContinuationPrefix ()

void chgContinuationPrefix (fmt)
char *fmt ;

int getVisible ()

int setVisible (newval)

int newval ()

Description

The expandLog function takes a compressed log message as produced by log and expands it to the human readable form using the textLogFmt file produced by lComp. It returns a pointer to a buffer containing the expanded message.

Its behavior is controlled by a number of additional routines. setExpandFmt changes the name of the expansion format file to *file*. endExpandFmt causes the current expansion format file to be closed. getExpandFmt causes the expansion format specified by *index* to be read in and a pointer to the expansion format is returned.

Message expansion is controlled by two different formats. There is a high-level format, which specifies what pieces of the message to print and

how and a specific format for the information section of the message. The standard message consists of the following parts:

Priority The priority of the message, which can be printed in one of two forms, a 2 character string or as a decimal digit. The default is a two character string.

Time The time of day, which is normally printed in the same format the routine ctime produces minus the final \n. There is a great deal of flexibility in printing the time. All printing options supported by the dateFONT? command.

Name The name of the process

Source The name of the source file where the message was generated

Line The line in the source file where the message was generated.

Message The text of the message itself, whose text is the combination of the compressed data and the message format from the expansion text file

The default format is:

```
%P %T %N %S:%L\n%M
```

The *%P and %T* specifiers can take arguments enclosed in ()s.

%P(%s) is the default and specifies that the priority be printed as a two character string. *%P(%d)* specifies that the priority be printed as a decimal digit.

%T() takes the standard date command options,

%m	The month of the year, 01–12
%d	The day of the month, 01–31
%y	The last 2 digits of the year, 00–99
%D	The date as mm/dd/yy
%H	The hour, 00–23
%M	The minutes, 00–59
%S	The seconds, 00–59
%T	The time of day as HH:MM:SS
%j	The day of the year, 001–366
%w	The day of the week, 0–6, with Sunday == 0
%a	The day of the week as a three letter abbreviation
%h	The month as a three letter abbreviation
%r	The time of day in HH:MM:SS AM/PM notation
%n	A newline character
%t	A tab character
%%	The '%' character

Any other characters appearing between the ()s to the *%T* specifier are printed as is.

getExpansionFmt gets a pointer to the current expansion format. chgExpansionFmt changes the current expansion format to that specified by *fmt*.

Normally, long lines are printed as is, but it is possible to request wrapping of long lines at a specific column and the continuation lines can be prepended with a specific prefix, if desired. getExpansionCutoff returns the current column at which wrapping takes place. If the value is 0, no wrapping is being performed. chgExpansionCutoff changes the column cutoff to *newcol*. getContinuationPrefix returns a pointer to the current continuation line prefix. The default is none. chgContinuationPrefix changes the continuation line prefix to that specified by *fmt*.

Normally all characters are printed as is. It is possible to request that control and nonprinting characters be made visible. getVisible returns 0 if control and nonprinting characters are not being made visible, which is the default. Currently any non-0 value indicates that the control and nonprinting characters are being converted to visible strings. setVisible sets the printing characteristics of control and non-printing characters to *newval*.

Environment Variables

The following environment variables are checked once, the first time expandLog is entered, if the parameter they specify has not already been set by the application developer.

LOGFORMAT A string specifying the printing format to be used

LOGCOLUMN The column at which line wrapping should take place

LOGCONTPREFIXThe string to be prepended to continuation lines

LOGROOT The directory in which textLogFmt is to be found if the expansion file is not specified otherwise

Caveats

Expansion is performed into a single allocated buffer. If more than one expansion is done, it is the responsibility of the caller to copy the expanded message from the buffer if the previous expansion is to be retained while a new expansion is being done.

Line wrapping and making control characters visible are both performed after the basic message expansion. They require additional manipulations of the message buffer and cost machine cycles.

See Also

lComp
logCat

logDstPri

Name

createLDParray, getLDPpriority, getLDPdsts, readDstPri, fmtLDPdsts, writeDstPri, freeLDPcontents, freeLDParray, indexLDPelement, deleteLDPelement, copyLDPcontents, insertLDPelement, replaceLDPelement — Routines to read, write, and manipulate the logging system's destination/priority assignment file

Synopsis

```
#include log.h
#include logDstPri.h

int createLDParray ()

int getLDPpriority ( name,priIndex,dpDA )
char *name,
int priIndex,
int dpDA

struct DstTranslator *appendDstTranslator ( dtp,name,index )
struct DstTranslator *dtp ;
char *name ;
int index ;

struct DstTranslator *mkDstTranslator ( dpDA )
int dpDA ;

void freeDstTranslator ( dtp )
struct DstTranslator *dtp" ;

int getLDPdsts ( spec,dtp,dstp )
char *spec ;/* Destination specification */
struct DstTranslator *dtp ;
unsigned int *dstp ;/* Where to return the destination mask */
```

```
int readDstPri ( name,fp,errp,checkFlag )
char *name“;/* Name of source - used only in messages */
FILE *fp ;/* Source of input */
int *errp ;/* Pointer to error count */
int checkFlag ;/* If TRUE, generate error messages about errors */



char *fmtLDPdsts ( dst,dtp )
unsigned int dst ;
struct DstTranslator *dtp ;

int writeDstPri ( dpDA,fpout )
int dpDA ;/* Dynamic array of LDPelement structures */
FILE *fpout; /* Where output written */

void freeLDPcontents ( lp )
struct LDPelement *lp ;

void freeLDParray ( dpDA )
int dpDA ;

indexLDPelement ( dpDA,lp )
int dpDA ;/* Dynamic array of structures */
```

```
struct LDPelement *lp ; /* Ptr to element to be deleted */

void deleteLDPelement ( dpDA,index )

int dpDA ;/* Dynamic array of structures */
int index ; /* Index of structure in array */

int copyLDPcontents ( lpDst,lpSrc )
struct LDPelement *lpDsp ;
struct LDPelement *lpSrc" ;

struct LDPelement *insertLDPelement" ( dpDA,pos,lp )
int dpDA ;/* Dynamic array descriptor */
int pos ;/* Position in array at which to insert structure. */
struct LDPelement *lp ;/* Structure to be inserted. */

struct LDPelement *replaceLDPelement ( lpDst,lpSrc )
struct LDPelement *lpDst ;
struct LDPelement *lpSrc ;
```

Description

createLDParray ( ) creates an initial dynamic array to hold the *LDPelement* structures. Normally it would not be called directly until a new file is being created from scratch. readDstPri ( ) calls it when it reads in an existing rules file.

readDstPri ( ) reads in a file containing the message priority and destination information. (See msgRules (4x) for details.) *name* is the name of the file containing the information. It is used only to report errors correctly. *fp* is an open standard I/O stream descriptor from which the information is read. *errp* is a pointer to an integer in which the number of errors detected during the reading process are returned. *checkFlag* causes any errors detected to produce an error message on the standard error stream in addition to an increment to the error count pointed to by *errp*.

writeDstPri ( ) writes out the information in the dynamic array described by dpDA. The information written to the standard I/O stream *fpout*. A FAILURE (-1) is returned if nothing can be written otherwise the number of records written is returned.

freeLDParray ( ) is the means to release the resources allocated by readDstPri ( ) or the other routines can add things to the dynamic array of LDPelement structures. When it returns, the dynamic array is closed and all the resources returned to the system.

indexLDPelement ( ) converts a LDPelement structure pointer into the index of the structure within the dynamic array of structures specified by dpDA. A FAILURE (-1) is returned if the array does not exist, *lp* does not point to a structure within the array of structures, or if *lp* does not point to the beginning of a structure within the array. This index value can be used with the deleteLDPelement ( ) routine to delete structures.

deleteLDPelement ( ) deletes the one specific LDPelement structure specified by *index* from the dynamic array of LDPelement structures specified by dpDA. All the resources associated with the structure are released and all structures above it are moved down one to fill in the hole.

insertLDPelement ( ) inserts a new LDPelement structure into the dynamic array specified by dpDA at the location specified by *pos*. If *pos* is negative or greater than the current size of the dynamic array, the new structure is inserted at the end of the current array. If *pos* is greater than or equal to 0 and less than the size of the array, then the new structure is inserted at that location in the array. The information for the new structure is pointed to by *lp*. No assumptions are made about the data in the structure. All strings have new copies made, hence it is the responsibility of the calling routine to release any data storage associated with *lp* as new copies of all data are made during the insertion process.

replaceLDPelement ( ) replaces the information in the LDPelement structure specified by *lpDst* with that pointed to by *lpSrc*. The original information used by *lpDst* is released before the replacement. As with insertLDPelement ( ), all data is copied. No pointers to data are reused, hence it is safe to release the data areas in *pSrc* once the replaceLDPelement ( ) is complete.

copyLDPcontents ( ) performs the same job as replaceLDPelement ( ) except that it assumes that *lpDst* points to an uninitialized LDPelement structure. It first zeros the structure and then makes copies of the information found in *lpSrc*.

freeLDPcontents ( ) releases all allocated strings used by the LDPelement structure pointed to by *lp*. Nothing is done with the structure itself.

getLDPpriority ( ) converts the ASCII representation of the priority specified by *name* into a priority value. It understands the predefined list of names E_NONE, E_MANUAL, E_MINOR, E_MAJOR, and E_CRITICAL as well as the numbers 0-4. It can additional understand the values specified by a *$priorities* specification if dpDA is a dynamic array descriptor for an array of LDPelement structures and *priIndex* is the index of the *$priorities* specification to be used for the translation. A FAILURE (-1) is returned if the *name* is not understood, otherwise a value from 0-4 is returned.

The getLDPdsts ( ) and fmtLDPdsts ( ) functions require a *DstTranslator* structure to operate. This structure contains to parallel arrays of names and index values. The structure can be created one destination element at a time with appendDstTranslator ( ). *name* is the ASCII name of the destination, for example, “log” or “console”. *index* is the index of the bit position (from 0–31) (and the index of entry describing this destination in *$LOGROOT/Config*). If *dtp* is NULL, a new DstTranslator structure is allocated and initialized. *name* and *index* are added to the appropriate arrays within this structure. The address of the DstTranslator structure is returned. NULL is returned if *name* does not point to a non-empty string or if index is not within the range 0–31. NULL is also returned if the allocation of a new DstTranslator structure fails. It is the responsibility of the caller to release the DstTranslator structure when it is no longer needed by calling freeDstTranslator ( ).

mkDstTranslator ( ) makes a complete DstTranslator structure from the dynamic array of LDPelement structures specified by dpDA. This is the type array returned by readDstPri ( ).

getLDPdsts ( ) converts an ASCII specification of destinations into a bit mask. The translation information needed to convert ASCII destination names into bits is supplied by the DstTranslator structure specified by *dtp*. getLDPdsts ( ) returns FALSE if it is unable to convert the *spec* properly and TRUE if the translation was successful. The value of the translated mask is returned in the unsigned int pointed to by *dstp*. The matching of ASCII destination names to those supplied by the DstTranslator structure is via the shortest unique match, hence the destination names need not be completely spelled out as long as they are unique.

fmtLDPdsts ( ) converts a bit mask into an ASCII representation of the legal destinations. *dst* is the bit mask of destinations. *dtp* is a pointer to a DstTranslator structure containing the mapping of bits to names. Each destination is separated by a '|' character and any bits not specified by the DstTranslator structure are printed as a decimal index of the bit (0–31).

See Also

expandLog(3x)
logCat(1x)
log(4x)
logMsg(3x)

logMsg

Name

logMsg, vlogMsg, logSysError, logInit — Log messages using the dynamic destination/priority mechanism

Synopsis

```
#include <varargs.h>
#include "log.h"
#include "logDstPri.h"

int logInit(procName)
char *procName;/* Name of sending process */

char *logMsg(msgNum,EL_FL,...)
EL_FL           /* Macro of __FILE__,__LINE__ */
int msgNum;/* Index number of loggin message */

char *vlogMsg(msgNum,El_FL,argPrt)
int msgNum;   /* Index number of loggin message */
EL_FL           /* Macro of __FILE__,__LINE__ */
va_list argPrt;  /* Pointer to arguments for loggin message */

char *logSysError(EL_FL,fmt,...)
EL_FL           /* Macro of __FILE__,__LINE__ */
char *fmt;
```

Description

logInit() is a simpler form of the loginit() function. It assumes that you will be using the logMsg(), vlogMsg(), or logSysError() routines and hence are not concerned with the values of the default priority or destination, what are arguments provided to loginit(). It only requires the name of the process as you want it to appear in the messages that are logged by the calling process. It specifies the default priority as E_NONE and the default destination as MASTER_LOG.

The routines logMsg(), vlogMsg(), and logSysError() provide a means to log messages with the PRISM logging system. Starting from the msgID supplied, a message priority and destination mask is looked up. This lookup information is stored in shared memory by the process logDstPri, which reads the file msgDst.rules from the directory /usr/spool/log, where the priority and destination masks are ultimately defined.

logMsg() log the message identified by msgNum to the logging system. The *EL_FL* macro identifies the place in the code where the call was generated. It is a macro expanding to *__FILE__,LINE__*. Any arguments required by a specific logging message format are provided after *EL_FL*.

vlogMsg() does the same job as logMsg(), but it takes any additional arguments from *argPrt*, which points to the arguments required by the specified message.

logSysError() generates an error message based on the current value of the global errno, which is set by the UnixWare system whenever a system call fails.

See Also

arrays
expandLog
fixLogFile
LComp
log
logDstPri

Numerics

23B+D

23 bearer (communication) and 1 data (signaling) channel on a T1 PRI circuit card.

30B+D

30 bearer (communication) and 1 data (signaling) channel (plus framing channel 0) on an E1 PRI circuit card.

47B+D

47 bearer (communication) and 1 data (signaling) channel on two T1 PRI circuit cards.

4ESS®

A large Lucent central office switch used to route calls through the telephone network.

5ESS®

A Lucent electronic switching machine used to route calls through the telephone network or private branch exchange.

A

AC

alternating current

ACD

automatic call distributor

AD

application dispatch

AD-API

application dispatch application programming interface

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, the Adjunct/Switch Application Interface) that the system administers via cut-through access to the inherent management capabilities of the product itself; this is in opposition to the ability of the system to administer the switch directly.

ADPCM

adaptive differential pulse code modulation

ADU

asynchronous data unit

advanced speech recognition

A speech recognition ability that allows the system to understand WholeWord and FlexWord™ inputs from callers.

affiliate

A business organization that Lucent controls or with which Lucent is in partnership.

AGL

application generation language

ALERT

System alerter process

alerter

A system process that responds to patterns of events logged by the “logdaemon” process.

American Standard Code for Information Interchange

A standard code for data representation that 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.

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.

ANI

automatic number identification

announcement

A message the system plays to the caller to provide information. The caller is not asked to give a response. Compare to prompt.

API

Application programming interface

application

The automated transaction (interactions) among the caller, the voice response system, and any databases or host computers required for your business.

application administration

The component of the system that provides access to the applications currently available on your system and helps you to manage and administer them.

application verification

A process in which the system verifies that all the components needed by an application are complete.

ASCII

American Standard Code for Information Interchange

ASI

analog switch integration

ASR

advanced speech recognition

asynchronous communication

A method of data transmission in which bits or characters are sent at irregular intervals and spaced by start and stop bits rather than by time. Compare to synchronous communication.

asynchronous data unit

An electronic communications device that allows computer systems to communicate over asynchronous lines more than 50 feet (15 m) in length.

automatic call distributor

That part of a telephone system that recognizes and answers incoming calls and completes these calls based on a set of instructions contained in a database. The ACD 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

Using a utility that makes an archive copy of a completed application or an interim copy of an application in progress. The back-up copy can be restored to the system if the on-line version is damaged, or if you make revisions and want to go back to the previous version.

barge-in

A capability provided by WholeWord and FlexWord speech recognition and Dial Pulse Recognition (DPR) that allows callers to speak or enter their responses during the prompt and have those responses recognized (similar to the Speak with Interrupt capability). See also echo cancellation.

batch file

A file containing one or more lines, each of which is a command executable by the UNIX shell.

BB

bulletin board

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.

bps

bits per second

BRDG

call bridging process

bridging

The process of connecting one telephone network connection to another over the system 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.

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), which is the equivalent of one character of text.

C

call classification analysis

A process that enables application designers to use information available within the system to classify the disposition of originated and transferred calls. CCA is an optional feature package.

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 the person making a telephone call. Telephone switching equipment can use this number to selectively route an incoming call to a particular department or agent.

caller

The party who calls for a service, gets connected to the system, and interacts with it. As the system can also make outbound calls for service, the caller can also be the person who responds to those outbound calls.

call flow

See transaction.

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 hardware 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.

CAS

channel associated signalling

caution

An admonishment or advisory statement used in the system documentation to alert the user to the possibility of a service interruption or a loss of data.

CCA

call classification analysis

CDH

call data handler process

CELP

code excited linear prediction

central office

An office or location in which large telecommunication devices such as telephone switches and network access facilities are maintained. These locations follow strict installation and operation requirements.

central processing unit

See processor.

CGEN

Voice system general message class

channel

See port.

channel associated signaling

A type of signaling that can be used on E1 circuit cards. It occurs on channel 16.

circuit card upgrade

A new circuit card that replaces an existing card in the platform. Usually the replacement is an updated version of the original circuit card to replace technology made obsolete by industry trends or a new system release.

cluster controller

A bisynchronous interface that provides a means of handling remote communication processing.

CO

central office

code excited linear prediction

A means of encoding analog voice signals into digital signals that provides excellent quality with use of minimum disk space.

command

An instruction or request the user issues to the system software to make the system perform a particular function. An entire command consists of the command name and options.

configuration

The arrangement of the software and hardware of a computer system or network. The system configuration includes either a standard or custom processor, peripheral equipment (for example, printers and 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 system 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 SSP and E1/T1 circuit cards, and perform various maintenance functions.

connect and disconnect (C and D) tones

DTMF tones that inform the system when the attendant has been connected (C) and when the caller has been disconnected (D).

connected digits

A sequence of digits that the system can process as a group, rather than requiring the caller to enter the digits one at a time.

controller circuit card

A circuit card used on a computer system that controls its basic functionality and makes the system operational. These circuit 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.

CPE

customer provided equipment or customer premise equipment

CPN

called party number

CPT

call progress tones

CPU

central processing unit

CPU Complex

The processor for the UCS 1000 R4.2 consisting of a single-board computing circuit card and an I/O companion board (SBC/IOB). The CPU complex is also used in other compactPCI platforms.

crash

An interactive utility for examining the operating system core and for determining if system parameters are being exceeded.

CSU

channel service unit

custom speech

Unique words or phrases to be used in system voice prompts that Lucent Technologies custom records on a per-customer basis.

custom vocabulary

A specialized package of unique words or phrases created on a per-customer basis and used by WholeWord or FlexWord speech recognition.

CVS

converse vector step

D

danger

An admonishment or advisory statement used in system documentation to alert the user to the possibility of personal injury or death.

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 record

The information in a database for a person, product, event, etc. The database record is made up of individual fields for each information item.

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 store permanently, in the transaction outline.

dB

decibel

DB

database

DBC

database checking process

DBMS

database management system

DC

direct current

DCE

data communications equipment

DCP

digital communications protocol

debug

The process of locating and correcting errors in computer programs; 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 a bus or on Tip/Ring, E1/T1, or SSP circuit cards.

dial ahead

The ability to collect and process touch-tone inputs in sequence, even when they are received before the prompts.

dial pulse recognition

A method of recognizing caller pulse inputs from a rotary telephone.

dialed number identification service

A service that allows incoming calls to contain information about the telephone number for which it is destined.

dial through

A capability provided by touch-tone and dial pulse recognition that allows callers to enter their responses during the prompt and have those responses recognized (similar to the Speak with Interrupt capability). See also barge-in and echo cancellation.

DIO

disk input and output process

DIP

data interface process

directory

A type of file used to group and organize other files or directories.

DMA

direct memory address

DNIS

dialed number identification service

DPR

dial pulse recognition

DSP

digital signal processor

DTE

data terminal equipment

DTMF

dual tone multi-frequency

DTR

data terminal ready

dual tone multi-frequency

A touch-tone sound that is an audio signal including two different frequencies. *DTMF feedback* is the process of the “switch” providing this information to the system. *DTMF muting* is the process of ignoring these tones (which might be simulated by human speech) when they are not needed for the application.

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 when the system crashes. The dump can be fetched after rebooting to help in analyzing the cause of the crash.

E

E&M

Ear and Mouth

E1 / T1

Digital telephony interfaces, commonly called *trunks*. E1 is an international standard at 2.048 Mbps. T1 is a North American standard at 1.544 Mbps.

Ear and Mouth

A common T1 trunking protocol for connection between two “switches.”

EBCDIC

Extended Binary Coded Decimal Interexchange Code

echo cancellation

The process of making the channel quiet enough so that the system can hear and recognize WholeWord, FlexWord, and dial pulse inputs during the prompt. See also barge-in.

editor system

A system that allows speech phrases to be displayed and edited by a user.

EIA

Electronic Industries Association

EISA

Extended Industry Standard Architecture

EMI

electromagnetic interference

Enhanced Basic Speech

Pre-recorded speech available from Lucent Technologies in several languages. Sometimes called standard speech.

error message

A message on the screen indicating that something is wrong with a possible suggestion of how to correct it.

ESD

electrostatic discharge

ESS

electronic switching system

EST

Enhanced Software Technologies, Inc.

ET

error tracker

Ethernet

A name for a local area network that follows IEEE standard 802.3. Supported implementations are 10BaseT and/or 100BaseT.

event

The notification given to an application when some condition occurs that is generally not encountered in normal operation.

EXTA

external alarms feature message class

external actions

Specific predefined system tasks that Script Builder can call or *invoke* to interact with other products or services. When an external action is invoked, the systems displays a form that provides choices in each field for the application developer to select. Examples are Call_Bridge, Make_Call, SP_Allocate, SR_Prompt, etc. In Voice@Work, external actions are treated as external functions.

external functions

Specific predefined (or customer-created) system tasks that Voice@Work or Script Builder can call or *invoke* to interact with other products or services. The function allows the application developer to enter the argument(s) for the function to act on. Examples are concat, getarg, length, substring, etc. See also external actions.

F

FCC

Federal Communications Commission

FDD

floppy disk drive

feature

A function or capability of a product or an application within the system.

feature package

An optional package that may contain both hardware and software resources to provide additional functionality to a standard system.

feature_tst script package

A standard system software program that allows a user to perform self-tests of critical hardware and software functionality.

FEP

front end processor

field

See database field.

FIFO

first-in-first-out processing order

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 (FTS).

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 in a specific language. See also subword technology.

foos

facility out-of-service state

FTS

file transfer process message class

function key

A key, labeled F1 through F8, on your keyboard to which the system software gives special properties for manipulating the user interface.

G

GEN

PRISM logger and alerter general message class

grammar

The inputs that a recognizer can match (identify) from a caller.

GUI

graphical user interface

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), while the existing platform and other existing optional circuit cards remain.

HDD

hard disk drive

hwoos

hardware out-of-service state

Hz

Hertz

IBM

International Business Machines

iCk or ICK

The system integrity checking process.

ID

identification

IDE

integrated disk electronics

idle channel

A channel that either has no owner or is owned by its default owner and is onhook.

IE

information element

IEEE

Institute of Electrical and Electronic Engineers

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.

independent software vendor

A company that has an agreement with Lucent Technologies to develop software to work with the system to provide additional features required by customers.

indexed table

A table that, unlike a nonindexed table, can be searched via a field name that has been indexed.

industry standard architecture

A PC bus standard that allows processors and other circuit cards to communicate with each other.

INIT

voice system initialization message class

initialize

To start up the system for the first time.

inserv

in-service state

Integrated Services Digital Network

A network that provides end-to-end digital connectivity to support a wide range of voice and data services.

intelligent CCA

Monitoring 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. The monitoring capabilities are dependent on the network interface circuit card and protocol used.

interface

The access point of a system. With respect to the system, the interface is designed to provide you with easy access to the software capabilities.

interrupt

The termination of voice and/or telephony functions when some condition occurs.

Intuity Response Application Programming Interface

A library of commands that provide a standard development interface for voice-telephony applications.

IOB

I/O companion card to the SBC. This is part of the CPU Complex.

IPC

interprocess communication

IPC

intelligent ports card (IPC-900)

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

K

Kbps

kilobytes per second

Kbyte

kilobyte

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 system to recognize a single word in the middle of an entire phrase spoken by a caller in response to a prompt.

L

LAN

local area network

LDB

local database

LED

light-emitting diode

library states

The state information about channel activities maintained by the IRAPI.

LIFO

last-in-first-out processing order

line side E1

A digital method of interfacing a system to a PBX or “switch” using E1-related hardware and software.

line side T1

A digital method of interfacing a system 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 LAN provides communications between computers and peripherals.

local database

A database residing on the system.

LOG

System logger process message class

logdaemon

A UNIX system information and error logging process.

logger

See logdaemon.

logging on/off

Entering or exiting the system software.

LSE1

line side E1

LSPS II

Lucent speech processing solutions II circuit card (6UB5)

LST1

line side T1

Lucent speech processing solutions II circuit card (6UB5)

A high-performance speech processing circuit card capable of simultaneous support for various speech technologies. In addition to the basic speech-processing features, The LSPS II circuit card provides enhanced Text-to-Speech capabilities and subword recognition for large vocabularies.

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 system screen from which you are able to enter either the System Administration or Voice System Administration menu.

maintenance process

A software process that runs temporary diagnostics and maintains the state of circuit cards and channels.

manoos

manually out-of-service state

masked event

An event that an application can ignore (that is, the application can request not to be informed of the event).

master

A circuit card that provides clock information to the TDM bus.

Mbps

megabits per second

MByte

megabyte

megabyte

A unit of memory equal to 1,048,576 bytes (1024 x 1024). It is often rounded to one million.

menu

Options presented to a user on a computer screen or with voice prompts.

MF

multifrequency

MHz

megahertz

ms

millisecond

msec

millisecond

MS-DOS

A personal computer disk operating system developed by the Microsoft Corporation.

MTC

maintenance process

multichannel application

A single process/application that controls several channels. Each channel of the application is managed explicitly. Typically this means state information for each channel is maintained and the state of the application on each channel is tracked.

multifrequency

Dual tone digit signalling (similar to DTMF), used for trunk addressing between network switches or by network operators.

N

NCP

Network Control Program

NEBS

Network Equipment Building Standards

NEMA

National Electrical Manufacturers Association

netoos

network out-of-service state

NFS

network file sharing

NM-API

Network Management - Application Programming Interface

NMVT

network management vector transport

nonex

nonexistent state

nonindexed table

A table that can be searched only in a sequential manner and not via a field name.

nonmasked event

An event that must be sent to the application. Generally, an event is nonmaskable if the application would likely encounter state transition errors by trying to mask it.

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

OEM

original equipment manufacturer

on-line help

Messages or information that appear on the user's screen when a function key (usually F1) is pressed.

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 executes 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

private branch exchange

PC

personal computer

PCB

printed circuit board

PCI

peripheral component interconnect

PCI Mezzanine Card

A PCI module, such as a LAN or RAID controller, that connects to the CPU Complex IOB companion card.

PCM

pulse code modulation

PEC

price element code

peripheral (device)

Equipment such as printers or terminals that is in addition to the basic processor.

peripheral component interconnect

A newer, higher speed PC bus that is gradually displacing ISA for many components.

permanent process

A process that starts and initializes itself before it is needed by a caller.

phoneme

A single basic sound of a particular spoken language. For example, the English language contains 40 phonemes that represent all basic sounds used with the language. The English 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 (screening)

The rejection of unrecognized speech. The WholeWord and FlexWord speech recognition packages can be programmed to reprompt the caller if the system does not recognize a spoken response.

phrase tag

A string of up to 50 characters that identifies 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.

pluggable

A term usually used with speech technologies, in particular standard speech, to indicate that a basic algorithmic technique has been implemented to accept one or more sets of parameters that tailors the algorithm to perform in one or more languages.

PMC

PCI Mezzanine Card

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.

polling

A network arrangement whereby a central computer asks each remote location whether it wants 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.

PRI

Primary Rate Interface

Primary Rate Interface

An ISDN term for connections over E1 or T1 facilities that are usually treated as trunks.

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 system documentation, the computer on which UnixWare and system software runs. In general, the part of the computer system that processes the data. Also known as the central processing unit.

prompt

A message played to a caller that gives the caller a choice of selections in a menu and asks for a response. Compare to announcement.

pseudo driver

A driver that does not control any hardware.

PSTN

public switch telephone network

pulse code modulation

A digital modulation method of encoding voice signals into digital signals. See also adaptive differential pulse code modulation.

R

RAID

redundant array of independent disks

RAID Array

An assembly of disk drives configured to provide some level of RAID functionality

RAM

random access memory

RDMBS

ORACLE relational database management system

RECOG

speech recognition feature message class

recognition type

The type of input the recognizer can understand. Available types include touch-tone, dial pulse, and Advanced Speech Recognition (ASR), which includes WholeWord and FlexWord speech recognition.

recognizer

The part of the system that compares caller input to a grammar in order to correctly match (identify) the caller input.

record

See database record.

recovery

The process of using copies of the system software to reconstruct files that have been lost or damaged. See also restore.

remote database

Information stored on a system other than your current system that can be accessed by your current system.

REN

ringer equivalence number

reports administration

The component of a system that provides access to system reports, including call classification, call data detail, call data summary, message log, and traffic reports.

restore

The process of recovering lost or damaged files by retrieving them from available back-up 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 using a component from a source system in a target system after a software upgrade or platform migration.

RFS

remote file sharing

RM

resource manager

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.

RTS

request to send

S

SCA

single connector architecture

SBC

A single-board computing circuit card used in the UCS 1000 R4.2. It is part of the CPU complex.

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 system 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 system (see also Voice@Work).

SCSI

small computer system interface

SDN

software defined network

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.

SID

station identification

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.

SLIP

serial line interface protocol

small computer system interface

A disk drive control technology in which a single SCSI adapter circuit card plugged into a PC slot is capable of controlling as many as seven different hard disks, optical disks, tape drives, etc.

SNA

systems network architecture

SNMP

simple network management protocol

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 system software.

software upgrade

The installation of a new version of software in which the existing platform and circuit cards are retained.

source system

The system from which you are upgrading (that is, your system as it exists *before* you upgrade).

speech and signal processor circuit card (CWB1)

A high-performance signal processing circuit card capable of simultaneous support for various speech technologies.

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.

speech modeling

The process of 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 space

An area that contains all digitized speech used for playback in the applications loaded on the system.

speech phrase

A continuous speech segment encoded into a digital string.

speech recognition

The ability of the system to understand input from callers.

SPIP

signal processor interface process

SPPLIB

speech processing library

SQL

structured query language

SR

speech recognition

SSP

speech and signal processor circuit card (CWB1)

standard speech

The speech package available in several languages containing simple words and phrases produced by Lucent Technologies for use with the system. 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 voice prompts and messages to the caller. This feature is also called Enhanced Basic Speech.

standard vocabulary

A standard package of simple word speech models provided by Lucent Technologies and used for WholeWord speech recognition. These phrases include the digits “zero” through “nine,” “yes,” “no,” and “oh,” or the equivalent words in a specific local language.

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 used in FlexWord recognition that recognizes phonemes or parts of words. Compare to WholeWord speech recognition.

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 (in other words, is *on hook*). The device is raised when the handset is picked up (in other words, when 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 system that enables you to define the interaction between the system 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 telephone 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. Compare to asynchronous communication.

SYS

UNIX system calls message class

sysgen

system generation

system administrator

The person assigned the responsibility of monitoring all system software processing, performing daily system operations and preventive maintenance, and troubleshooting errors as required.

system architecture

The manner in which the system software is structured.

system message

An event or alarm generated by either the system or end-user process.

system monitor

A component of the system that tests to verify that each incoming telephone line and its associated Tip/Ring or T1 circuit 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

See database table.

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.

talkoff

The process of a caller interrupting a prompt, so the prompt message stops playing.

TAM

telecom alarm module

target system

The system to which you are upgrading (that is, your system as you expect it to exist *after* you upgrade).

TAS

transaction assembler script

TCP/IP

transmission control protocol/internet protocol

TDM

time division multiplexing

telecom alarm module

An intelligent alarm module that provides critical, major, and minor alarm indicators.

telephone network connection

The point at which a telephone network connection terminates on a system. Supported telephone connections are Tip/Ring, T1, and E1.

Text-to-Speech

An optional feature that allows an application to play US English 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.

ThickNet

A 10-mm (10BASE5) coaxial cable used to provide interLAN communications.

ThinNet

A 5-mm (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

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

The interactions (exchanges) between the caller and the voice response system. A transaction can involve one or more telephone network connections and voice responses from the system. It can also involve one or more of the system optional features, such as speech recognition, 3270 host interface, FAX Actions, etc.

transaction assembler script

The computer program code that controls the application operating on the voice response system. The code can be produced from Voice@Work, Script Builder, or by writing directly in TAS code.

transaction state machine process

A multi-channel IRAPI application that runs applications controlled by TAS script code.

transient process

A process that is created dynamically only when needed.

troubleshooting

The process of locating and correcting errors in computer programs. This process is also referred to as debugging.

TSO

time share operation

TSM

transaction state machine process

TTS

Text-to-Speech

TWIP

T1 interface process

U

UCS

Unified Communications Server

UK

United Kingdom

US

United States of America

UNIX Operating System

A multiuser, multitasking computer operating system originally developed by Lucent Technologies.

UNIX shell

The command language that provides a user interface to the UNIX operating system.

upgrade scenario

The particular combination of current hardware, software, application and target hardware, software, applications, etc.

usability

A measurement of how easy an application is for callers to use. The measurement is made by making observations and by asking questions. An application should have high usability to be successful.

USOC

universal service ordering code

UVL

unified voice library

V

VDC

video display controller

vi editor

A screen editor used to create and change electronic files.

virtual channel

A channel that is not associated with an interface to the telephone network (Tip/Ring, T1, LSE1/LST1, 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 can be initiated only by a “virtual seizure” request to TSM from a DIP.

vocabulary

A collection of words that the system 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@Work

An optional software package that provides a graphical interface to assist in development of voice response applications on the system (see also Script Builder).

voice channel

A channel that is associated with an interface to the telephone network (T1, E1, or PRI). Any system application can run on a voice channel. Voice channel applications can 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 DIP or the **soft_szr** command.

voice processing co-marketer

A company licensed to purchase voice processing equipment 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 SSP circuit cards) and data storage devices (for example, hard disk, etc.)

voice response unit

A computer connected to a telephone network that can play messages to callers, recognize caller inputs, access and update a databases, and transfer and monitor calls.

voice system administration

The means by which you are able to administer both voice-related aspects of the system.

VPC

voice processing co-marketer

VROP

voice response output process

VRU

voice response unit

W

warning

An admonishment or advisory statement used in system documentation to alert the user to the possibility of equipment damage.

watchdog timer

An timer that activates a TAM alarm when CPU activity is not received within the 30-second threshold.

WholeWord speech recognition

An optional feature package based on whole-word technology that can recognize the numbers one through zero, “yes”, and “no” (the key words). This feature is reliable, regardless of the individual speaker. This feature can identify the key words when spoken in phrases with other words. A string of key words, called *connected digits*, can be recognized. During the prompt announcement, the caller can speak or use touch tones (or dial pulses, if available). See also whole-word technology.

whole-word technology

The ability to recognize an entire word, rather than just the phoneme or a part of a word. Compare to “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 available for FlexWord recognition by an application during a Prompt & Collect action step.

word spotting

The ability to search through extraneous speech during a recognition.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W