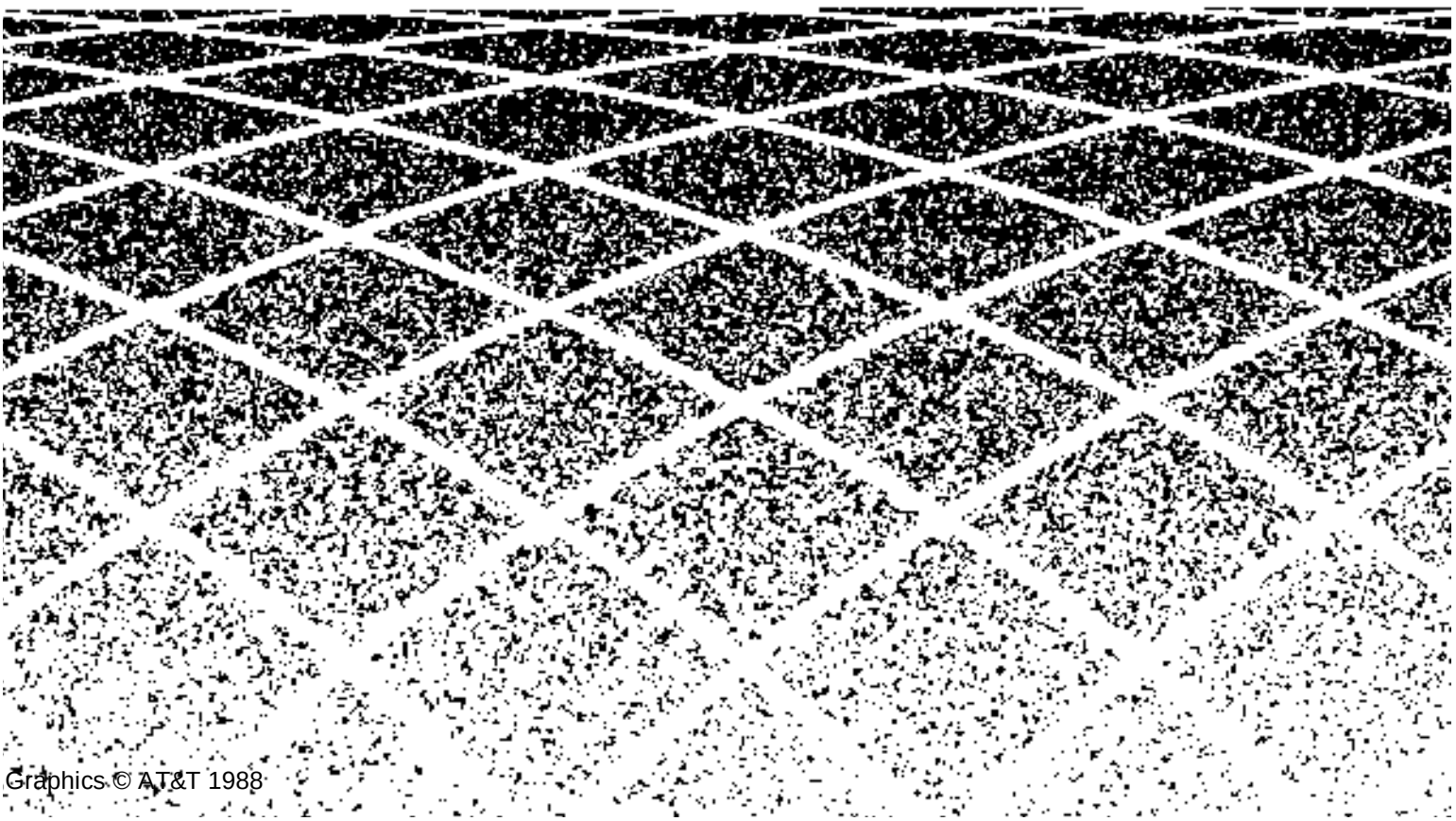
585-310-227
Issue 1
March, 1995

INTUITY Conversant Application Development



Blank Page

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

About This Book

Purpose

This book is a reference for people who develop applications for the AT&T Intuity™ CONVERSANT® Voice Information System (VIS) using the transaction state machine (TSM) script level language and/or C language. It provides information about designing software applications and writing programs that integrate the application software and generic software. Use this book by itself or in conjunction with the *Intuity CONVERSANT VIS V5.0 IRAPI Programming Guide*, 585-310-226, or the *Intuity CONVERSANT VIS V5.0 Script Builder,* 585-310-727.

How to Use This Book

The three activities you must complete for each application are:

- Design the application (Chapter 1, "Application Design" and Chapter 2, "Development Guidelines").
- Prepare the speech data that will be stored on disk (refer to *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228).
- Write the application programs (Chapter 3, "Script Instructions", Chapter 4, "Data Interface Process" and Chapter 5, "Adding and Modifying System Messages"). Every application requires a *script*. The VIS uses a proprietary script language (described in Chapter 3, "Script Instructions") to specify this application control logic. In addition, a data interface process (DIP), described in Chapter 4, "Data Interface Process", allows the script to communicate with a C language process for host and/or database access.

How This Book Is Organized

This book is organized into the following chapters:

- Chapter 1, "Application Design" 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, "Development Guidelines" provides an outline of the directory structure and naming conventions you should use when developing application programs.
- Chapter 3, "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 Process" 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, "Adding and Modifying System Messages" describes how to add or change system messages and it associated explain text.
- Chapter 6, "Upgrade Considerations" describes the procedures to upgrade an existing VIS environment to the current V5.0 environment to utilize logging capabilities.
- Chapter 7, "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 A, "Summary of Script Instructions" contains manual pages for each script instruction, including the syntax, arguments, and examples.
- Appendix B, "Voice System C-Library Functions" contains manual pages for each voice system C-library function, including the syntax, arguments, and examples.

This book also includes a list of abbreviations, a glossary, and a cross-referenced index.

Conventions Used in This Book

The following typographic conventions are used in this book:

- Terminal keys
  - — Terminal keys are shown in rounded boxes. For example, an instruction to press the enter key is shown as

    Press ENTER.

  - — Function keys (also known as *soft* keys) are shown in rounded boxes followed by the function of that key in parentheses. For example, an instruction to press function key 3 is shown as

    Press F2 (CHOICES).

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

    Press ALT D.

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

    Enter **y** to continue.

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

    Type **y** to continue.

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

  - — The word *select* is used to mean one of the following:

    a. Move to the desired menu item using the arrow keys and press ENTER. For example, an instruction to select an item from a menu and press ENTER is shown as

    Select Configuration Management from the Voice System Administration menu.

b. Type the first character of the item. The first menu item beginning with that letter is selected. If more than one item begins with the same letter, then type enough letters to identify the desired item. Press ENTER when the correct item is highlighted.

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

Enter **root** at the `Console Login` prompt.

— Command and file names and their parameters are shown in **bold** type. Variable parameters are shown in ***bold italic*** type when they are part of a user input and in *regular italic* type when they are not. All are illustrated in the following example:

Use the **print** command to print your report. The command syntax is **print *reportname***, where *reportname* is the name of the report to be printed.

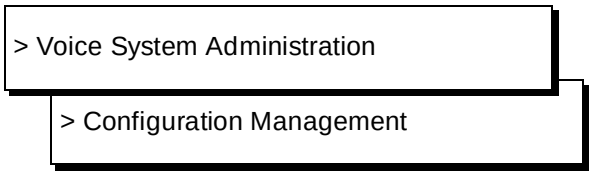
■ Screen displays

— Information that is displayed on your terminal screen — including screen displays, prompts, script code, and system messages — is shown in `typewriter-style` type; for example

```
Installation is in progress -- do not remove
the floppy disk.
```

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

Begin at the CONVERSANT Administration menu, and select the following sequence:



In this example, you would first access the CONVERSANT Administration menu. Then you would select the Voice System Administration option to display the Voice System Administration menu. From that menu, you would select the Configuration Management option to display the Configuration Management menu.

— The screens shown in the Intuity CONVERSANT library are only examples. Your screens may not appear exactly as illustrated.

Related Resources

The following books should be used in conjunction with this book:

- *Intuity CONVERSANT VIS V5.0 Script Builder,* 585-310-727 — This book describes the VIS Script Builder and how to use it to design applications to be used in the VIS environment. This book will be referred to when discussing relationships between Script Builder and TSM applications.
- *Intuity CONVERSANT VIS V5.0 Intuity Response API Programming Guide,* 585-310-226 — This book provides an description of the application development process using the Intuity Response Application Programming Interface (IRAPI), including the use of functions and processes through examples. This book will be referred to when discussing relationships between IRAPI and TSM applications.
- *Intuity CONVERSANT VIS V5.0 Application Design Handbook*, 585-310-551 — This book contains guidelines and suggestions for good application design for the VIS. This book will be referred to when discussing the process of designing, implementing, and deploying applications.
- *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228 — This book provides information concerning the speech development environment available on the VIS platforms. This book will be referred to when discussing speech editing systems as well as the creation and storage of speech data on the system.
- *Intuity CONVERSANT VIS V5.0 Communication Development*, 585-310-229 — This book provides information about creating the necessary platform environment and applications to get callers into the VIS environment and retrieve data for the caller through telephony and data network connections. This book will be referred to when discussing communication development issues of TSM scripted applications.

A full description of the Intuity CONVERSANT VIS library is available in the *Intuity CONVERSANT VIS V5.0 Documentation Guide,* 585-310-002.

Technical Updates

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

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

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

Trademarks d in This Book

The following trademarked products are mentioned in the document library:

- AUDIX, CONVERSANT, DEFINITY, 5ESS, and 4ESS are registered trademarks of AT&T.
- Voice Power, Intuity, and FlexWord are trademarks of AT&T.
- UNIX is a registered trademark of Novell, Inc.
- ORACLE, ORACLE*Terminal, OBJECT*SQL, SQL*FORMS, SQL*Menu, SQL*Net, SQL*Plus, PRO*C, and SQL*ReportWriter are trademarks of the Oracle Corporation.
- i486 is a trademark of the Microsoft Corporation.
- X Windows is a trademark of the Massachusetts Institute of Technology.
- Metro Link X is a registered trademark of Metro Link Corporation.
- Audio Works Station is a trademark of Bitworks® Inc.
- IBM is registered trademark of International Business Machines.
- CLEO and LINKix are trademarks of CLEO Communications.
- Hayes and Smartmodem are trademarks of Hayes Microcomputer Products, Inc.

How to Comment on This Book

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

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

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

Please include the name and number of this book.

Application Design

1

What's in This Chapter

This chapter describes, in general terms, points to consider when developing an application. Specific procedures for developing application programs are covered later in Chapters 3 through 5.

The first part of this chapter discusses issues involving the capabilities and limitations of the user and the Intuity CONVERSANT Voice Information System (VIS). The second part of this chapter recommends steps for application development.

TSM Applications vs. IRAPI Applications

This book details the design of transaction assembler (TAS) script applications. The Intuity Response Application Programming Interface (IRAPI) provides an alternative for developing applications using C-language. The transaction state machine (TSM) is written as an IRAPI application. For detailed information about the IRAPI, refer to the *Intuity CONVERSANT VIS IRAPI Programming Guide*, 585-310-226.

Designing a Successful Application

A successful application meets the following three criteria:

- The end user can easily access and use the service offered.
- The information provided for the end user is adequate.
- The connect time for each telephone call accessing the application is minimized.

To design an application to meet these criteria, you must consider the overall system including: the Intuity CONVERSANT VIS, the end user, and the data source (which may be a separate host computer).

End User Capabilities

Consider the following capabilities when developing your application for the end user:

- Short-term human memory
- Information processing

Short-Term Human Memory

Study results of short-term memory indicate the following:

- Limit the number of items a person is to remember to four.

  Write the dialog so the user is given a maximum of four items to remember For example, the following limits the number menu choices to four items.

  > To access your account balance, press 1. To request billing information, press 2. To schedule an appointment, press 3. To speak to a representative, press 0.

- Group items of more than four into logical sets.

  For example, group the number string 464764 into two, three-digit items as 464 and 764. This makes it easier for the user to remember a total of six digits.

- Place actions as close to the end of a spoken instruction as possible. For example:

  > Dial 431 to hear more about this special offer.
  > To hear more about this special offer, dial 431.

  The first phrase is more difficult for a user to remember than the second, because the call to action is in the first part of the sentence. The second example is much easier to remember because the call to action is at the end of the sentence.

Human short-term memory is susceptible to forgetfulness through interference. After an item, such as a menu number, is set in short-term memory, its likelihood of being forgotten increases as more auditory information is input.

Information Processing

The end user has a finite capacity to understand and to act on what is said by the VIS. When the end user is forced to spend more time trying to understand what is said, less information is remembered.

A well-designed transaction should have dialog that is intelligible to the end user at the individual word level and at the phrase level.

Natural rhythm and intonation help the user to understand the concatenated speech produced by the system. During the recording of speech, it is important that the professional speaker maintains consistent rhythm and intonation throughout the recording session (details in Chapter 2, "Development Guidelines" of *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228).

Make full use of the script instructions **tnum** and **tchar** to control the intonation of spoken numbers and letter/digit strings (described in Chapter 3, "Script Instructions"). Monitor the intelligibility of phrases as the application is developed.

Voice System Capabilities

The TAS script is executed by the transaction state machine (TSM) program (a UNIX system process) when it talks to end users and supplies the requested information. TSM reads the instructions written for a particular application (see Chapter 3, "Script Instructions").

The system capabilities that you must consider when designing an application are:

- Easily understood speech
- Recognition of touch-tone signals and spoken input

Easily Understood Speech

The end user hears speech as individual phrases that are spoken by the system. Processing speed eliminates any perceptible interval between the end of one phrase and the start of the next. As a result, individual phrases can be combined to form a complete sentence to be played to an end user. However, there are two problem areas to be aware of when combining phrases:

- Pitch and prosody (rhythm)
- Silence and pauses

Pitch and Prosody

Even if the same speaker is used to record a set of phrases, if there is a wide variation in the average pitch among individual phrases, then the concatenated phrases are difficult for the end user to process and remember. Variation in prosody (rhythm and emphasis) between different phrases also disturbs the listener.

During recording of the phrases, make sure the speaker maintains a steady pitch and rhythm; otherwise, concatenated phrases, such as those used in constructing strings of digits like telephone numbers, will sound stilted and artificial (see Chapter 2, "Development Guidelines" of *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228).

Rhythm and prosody are of special concern when the Text-to-Speech (TTS) feature is used. For further information on TTS, refer to Appendix D, “Advanced Text-to-Speech Features,” of *Intuity CONVERSANT VIS V5.0 Speech Development,* 585-310-228.

Silence and Pauses

Typically when *raw* phrase files are edited (see Chapter 3, “Editing Speech,” of *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228), silence is trimmed from the beginning and end of each phrase. When the resulting phrases are concatenated, the natural pause between the two is eliminated and the combined phrases can sound unnatural at the juncture point. This effect can be remedied by inserting silence or pauses of various durations where needed between concatenated phrases.

NOTE:
An application that has a large number of short phrases may experience performance problems in the form of speech breaks. Refer to Chapter 7, “Performance Information,” of *Intuity CONVERSANT VIS V5.0 System Description*, 585-310-225, for additional information on performances issues.

Issues with Touch-Tone and Speech Recognition

As a receiver for the end user’s input, the VIS accepts touch-tone input or speech. Touch-tone input is relatively straightforward: at a point in the transaction where the end user is to enter touch tones, the system listens (subject to time-out parameters described in Chapter 3, "Script Instructions") for a specified number of individual touch tones. The VIS can identify touch tones as accurately as the end user can input them. Issues to consider include type-ahead or touch-tone flushing capabilities.

Speech input is more complicated. Examples of considerations for speech input are:

- How the application should detect and correct errors
- How to define informative prompts for speech recognition
- How to recognize normal speaking habits that may cause problems (for example, “1040” may be spoken “ten-forty”)
- Whether to allow *barge-in* (interrupting the prompt with speech)
- How to avoid long strings that decrease the chance of correct speech recognition.

For more information on speech recognition, refer to Chapter 4, “Recognizing Speech Input,” of *Intuity CONVERSANT VIS V5.0 Speech Development,* 585-310-228.

System Interface with Host

You may need to define the interface between the VIS and a host computer. The definitions made at this point are used in designing the application and the logic for the data interface process (DIP) code (see Chapter 4, "Data Interface Process"). Before defining the host interface, answer the following questions:

- Is communication with a host needed?
- Are *I-am-alive*/host-check messages needed?
- Is data to be sent to or from the host as a batch job or on an as-needed basis or both?
- What type of database information is needed and how often it is needed?
- Will the VIS respond to host inquiries/requests or will it only initiate communications?
- How should the VIS respond if the host does not respond?
- How are the VIS local databases initialized?
- What should the VIS communicate to the host (if anything) when a new call comes in (maybe send a start-up message to the host)?
- What should the VIS do when it reboots?
- What should the VIS do when the host computer reboots?
- How much data is passed to/from the host computer at any time?

Determine Functional Data Requirements

The first step in writing the application is to determine the functional data that is to pass between the system and the end user. The second step is to determine the responses that the end user hears during the transaction.

To develop an application with efficient transactions, it is important to have a clear statement of the functional data that is to pass between the system and the end user and possibly a host computer. To ensure that the functional data requirements are complete and well-defined, this should be a joint effort between the developer, who knows the system capabilities, and the client, who knows the needs of the end user. Think of the application in the following manner: What kind of information does the user want from the system and what kind of information will the system require from the user?

Requirements can be determined in many ways. One effective approach is to view the transaction as an exchange of information between the end user and the VIS. This approach facilitates integration of dialog with the overall application development.

Develop Dialog

For information on developing scripts and prompts, refer to “Writing the Script” section in Chapter 3, “Developing Speech,” of *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228.

For each point in the dialog where a data transfer occurs, the dialog script must be expanded to account for all conceivable error states that might result. (The most time-consuming part of script development is trying to account for all error conditions, determine all potential causes, and then write dialog to cover these potential problems.)

Insert Hidden Housekeeping Routines

Housekeeping routines, such as counting and timing, can be inserted as needed to count the number of attempts made to access certain information objects, to record time required for database accesses, etc. These values may be stored in event memory (Chapter 3, "Script Instructions") and, at the termination of a call, are passed to the call data handler (CDH).

Test and Review Transaction

When the transaction program and its supporting routines have been written, the system should be tested, first with the client, and then with representative end users. A load test should be conducted during which the maximum number of callers access the application simultaneously. The objectives of this testing include verifying the transaction's operation and evaluating end users' acceptance. You must be prepared to revise the script based on feedback from testing until the transaction is judged to be acceptable from the end user's point of view.

Development Guidelines

2

What's in This Chapter

This chapter discusses the guidelines for developing transaction assembler (TAS) application programs for the Intuity CONVERSANT Voice Information System (VIS). The chapter provides a list of the basic tools that are available with the VIS, with guidelines 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 Programs

After you have documented the functional data requirements and the system responses, you are ready to begin working on the application programs. This work may begin even before the speech has been processed for access by the script.

Two programs must be developed for each application as described below.

- Script

  A script functions as a set of instructions used by the transaction state machine process (TSM) 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 an asynchronous host. You must write a DIP module in C-language. This requires an understanding of the UNIX operating system as well as C-language. Refer to Chapter 4, "Data Interface Process" for more information on writing a DIP.

  NOTE:
  The information in Chapter 4, "Data Interface Process" is provided for backward compatibility. All DIPs produced for use with Intuity CONVERSANT VIS V5.0 should be written in terms of the Intuity Response Application Programming Interface (IRAPI). Refer to the *Intuity CONVERSANT VIS V5.0 IRAPI Programming Guide*, 585-310-226, for additional information.

You can write a script using Script Builder or the script language described in Chapter 3, "Script Instructions" The Script Builder standard features are sufficient to support most applications.

If you find that you need capabilities not supported by Script Builder, you can write an external function using script language that is accessed by the Script Builder-generated application. For more information on writing external functions, refer to Chapter 12, “Using Advanced Features,” of *Intuity CONVERSANT VIS V5.0 Script Builder,* 585-310-727. If the external functions do not provide the capability required or if the application must access a database or perform complex calculations, you may write a DIP to perform these actions.

Application Development Tools

The standard set of tools available for the application developer include UNIX operating system tools, file processing programs tools, and debugging tools.

Figure 2-1 illustrates the typical steps in developing an application, specifically what tools to use at each step.

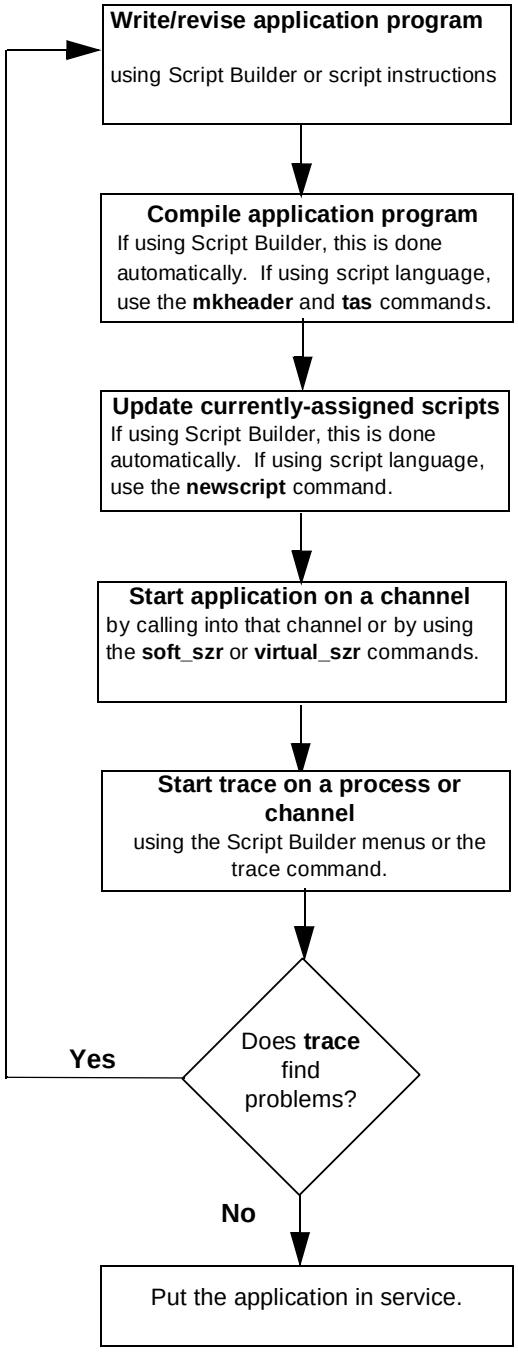


Figure 2-1. Using Application Development Tools — Example

Application Development Tools — Description

Most of these tools are described in detail in *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230. Some of these tools also are discussed in other chapters of this book or in other books. In these cases, the other chapters or books are noted.

- The speech administration tools (that is, **add**, **copy**, **erase**, **list**) are available for use with speech files stored in the default filesystem **/home2/vfs/talkfiles**.
- The **mkheader** command helps to define memory used by the TSM script (Chapter 3, "Script Instructions"). This program creates the ***application_name*_aloc.c** program and the ***application_name*def.h** file.
- The **newscript** command updates changes to all currently assigned scripts. If you write an application using script language and use tas to assemble that script, you must use newscript to insure that the most recent version of the script is used. If you develop the application using Script Builder, it is not necessary to use newscript because Script Builder automatically executes this command.
- The **soft_szr** command starts a script on a channel independent of an incoming call. If the script name is given a null string, TSM starts the script that is assigned to the channel specified in the software seizure request.
- The transaction assembler (**tas**) program accepts a file in script source code and produces a TSM executable file (Chapter 3, "Script Instructions").
- The **virtual_szr** command starts a script on a virtual channel.

Application Tracing Tools

The **trace** command activates a monitor process on specific DIPs and/or channels. In order for trace to be useful, however, you must place trace script instructions in the script and trace subroutines in the DIP.

The **trace** script instruction prints variables and status messages while the script is running and stores them in a buffer. The **trace** command displays this information. Refer to the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230, for more information about the trace command.

From a DIP process, you can use four subroutines to trace either a DIP or a specific channel to determine if there are problems with the application and, if there are any, where the problems are located. These processes are provided in the system library (**/vs/lib/libspp.a**). The output of these subroutines can be viewed using the trace command. Refer to Chapter 4, "Data Interface Process" for additional information.

The four subroutines that enable tracing are as follows:

NOTE:
The following subroutines are provided for backward compatibility. For information about the *irTrace(3IRAPI)* routines introduced in Intuity CONVERSANT VIS V5.0, refer to the *Intuity CONVERSANT VIS V5.0 IRAPI Programming Guide*, 585-310-226.

- **db_init** (whoami) — This subroutine is called once for set up. Whoami is the message origin name (for example, DIP0 and DIP2, which are defined in **/mesg.h**)
- **db_put** (str) — This subroutine is an unconditional send of the character string.
- **db_pr** (format, a1, a2, a3, a4,a5, a6) — This subroutine is a conditional send used to pass information to the tracing file only for tracing the calling process.
- **db_ch** (chan, format, a1, a2, a3, a4, a5, a6) — This subroutine is a conditional send used only if someone is tracing the identified channel.

For more information about the **trace** script instruction, refer to Appendix A, "Summary of Script Instructions."

Filesystem Organization

The following information details the current directory structure for Intuity CONVERSANT VIS V5.0 and precautions that should be taken to protect the information in those directories.

Directory Structure

The standard VIS large capacity fixed disk is divided into common filesystems such as **root**, **swap**, **usr**, and **home**. Each of these filesystems has many directories that are created and populated by the UNIX system software. Other directories are created and used by the VIS software.

Directory Usage Precautions

You should take the following precautions when using the VIS directories:

1. Backup excess or repetitive files to floppy disks, then delete them from the root file system when root space becomes low. If root runs out of space, the software could malfunction.
2. Backup files periodically. This ensures that if the files are damaged, you can recover the data.
3. Remove temporary files or copy these files to diskettes to maintain or create free space within a directory that has run out of free space.

Refer to the *Novell UnixWare Documentation Set*, 585-350-908, for information on backing up and managing system files.

Conventions for Naming Files and Programs

To make files easy to identify and to meet the requirements of the C compiler, the VIS 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. All the following files and programs in bold are provided by the application developer.

Table 2-1. File and Program Naming Conventions

File and/or Program	Description
***name*.c**	This identifies a C-language source program. For example, **hostmeas.c** or **msg1hdlr.c**
***name*.o**	This is a compiled C-program in which external references are not resolved. For example, **hostmeas.o** or **msg1hdlr.o**
***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. For example, **et.h** or **hwrtype.h**
DIP*N*	Each DIP is referenced by the name **DIP*N*** where *N* is a number or a word. Refer to Chapter 4, "Data Interface Process" for more information on DIPs. For example, **DIP0** or **DIP_test**
***application_name*.t**	This is the script source file for *application_name*. For example, **stock.t**
***application_name*.T**	This is a script that has been processed by using the **tas** command with ***application_name*.t** as an argument. For example, **stock.T**

Continued on next page

***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. For example, **stockdef.h**

Table 2-1. File and Program Naming Conventions — *Continued*

File and/or Program	Description
***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. For example, **stock_aloc.c**
***application_name*.D**	This file contains descriptions of application variables that normally are used as event counters. For example, **stock.D**

Continued on next page

infile.*application_name*	This file is created by the speech file developer. It lists each coded speech file and the associated ASCII phrase by which the files are identified by the script (when Script Builder is not used). For example, **infile.stock**
list.*application_name*	This file is created by the **numasgn** command (when Script Builder is not used). It consists of the **infile.*application_name*** file plus the talkfile number, an application identifying string, and phrase numbers. The **list.*application_name*** is given as the argument of the **tfile** script instruction (Chapter 3, "Script Instructions"). For example, **list.stock**
***application_name*.pl**	This is the talkfile created by Script Builder.

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, labels, inline comments, and **goto** script instructions.

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==im.MISTAKE, CORRECT)
```

Define statements can be put in the header files and included in the program by using C-language **#include** statements, which link the definitions to the program code during assembly or compilation. A define should be used only once for the same memory location. By convention, defines are in uppercase letters. They may have underscores (_) but no embedded spaces.

One file with a set of defines can be used for both the script and a DIP. This insures 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:

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

where *script.t* is the script source file and *1068* is the line in which the define appears. The limit to the number of define statements that a script may 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.

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 == im.0, strt_idloop )
...

strt_idloop:
getdig(0, 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 either should 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 == im.0, strt_idloop )
...
```

Script Instructions

3

What's in This Chapter

The conventions used in writing a script are described in this chapter, along with all the instructions needed to develop the application script. The instructions for writing a script are grouped under the following headings:

- Voice output
- Data gathering
- Data manipulation
- String manipulation
- Flow control
- Equipment allocation
- Voice coding
- Network interface
- Feature related
  - — Full Call Classification Analysis (CCA)
  - — Primary Rate Interface
  - — Text-to-Speech (TTS)
  - — WholeWord Speech Recognition
  - — FlexWord Speech Recognition
- Miscellaneous

Overview of TSM Actions

The transaction state machine (TSM) software process is an Intuity Response Application Programming Interface (IRAPI) application which manages the execution of transaction assembler (TAS) language application scripts. (See the *Intuity CONVERSANT VIS V5.0 IRAPI Programming Guide,* 585-310-226, for additional information about IRAPI). Application scripts are generated by using Script Builder or by using an editor to write scripts in the TAS instruction language and compiling them with the tas command.

The next few paragraphs discuss TSM actions at a lower level. If you are not interested in this level of detail, turn to "Call Progression."

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. Refer to Chapter 1, "Introduction to the IRAPI," in *Intuity CONVERSANT VIS V5.0 IRAPI Programming Guide*, 585-310-226, for a detailed description of the Intuity CONVERSANT VIS V5.0 system architecture.

For example, when TSM receives a **tflush** instruction from the script after phrases have been specified with the **talk** instruction, it uses the *irEnd()* IRAPI function to send an interprocess communication (IPC) message to the voice response output process (VROP). VROP uses the talkfile and phrase numbers specified by the **talk** instruction to tell the Tip/Ring (T/R)-T1 interface the phrases to play for a meaningful response to the caller. In the case of a **dbase()** instruction to a host computer, TSM communicates with a data interface process (DIP) through the *irPostEventQ()* IRAPI function. This process provides the interface between the generic host communication software and the application software by formatting the host messages between the two.

The script's assembly language-type instructions, running within the generic TSM software, are a sequence of library function calls that manage the low level interactions required to operate the system. There may be multiple invocations of the same script as well as the execution of several scripts concurrently within the TSM process. At any time 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 (refer to the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-350-230, for more information about **newscript**).

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 application and processes script functions through the IRAPI. TSM returns ownership of the channel to the AD process when the calls 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's data and sends a call data record to the call data handler (CDH), which makes it available in reports to the host and the VIS.

Call Progression

A T/R or T1 circuit card accepts a call made to the system by appropriate signaling to the central office or switch. The T/R circuit card then informs the Tip/Ring interface process (TRIP) or the T1 circuit card informs the T1 interface process (TWIP) that the call has been accepted. TRIP/TWIP informs the AD process that a new call has arrived. AD checks a service table to determine the script required for the call. In this case, TSM is the selected script. At this point, TSM reads the script instructions and lets the script control the sequence of events during a call. When the call has ended, TSM takes control.

Starting Conditions

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

- A caller dials and reaches the VIS's incoming telephone facility.
- The software recognizes the ringing condition.
- AD checks an internal table to determine what 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

At the point that all starting conditions have been met, the script takes control and typically executes the following functions during a transaction:

- 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
- Indicates its termination to TSM

TSM Control

When the script has ended, TSM takes control and performs the following functions:

- Puts the line on hook or terminates the script if one of the following conditions exists:
  - — 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. For example, it terminates if the last instruction, either a goto or a quit, is missing.
- 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
- Makes the channel available for the next call

Data Storage

The script has 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, "Development Guidelines" and *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230.

  Each script is allocated 512 bytes for user space but automatic allocations insure 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 is one hundred 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.

  r.0 (and occasionally r.1 and r.2) is a return register which may be used to indicate the results of a specific instruction. For example, the **dbase** instruction (described under “Script Instructions” in this chapter) 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 A, "Summary of Script Instructions"

Registers also may 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.

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 may store application data in event memory and any application-related data may be saved. 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 CDH. Call data is stored in the ORACLE database.

The reports generated from the database are available to the systems operator (refer to the Appendix B, “Database Environment,” of *Intuity CONVERSANT VIS V5.0 Operations,* 585-310-550).

The .D File

The **.D** file is created by Script Builder when you specify call data parameters. It also may be created manually if Script Builder is not used. 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 a valid event specification record. 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:

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

where *WS* refers to a tab or blank space and *STR* refers to the literal string STR.

The following is an example of the **.D** file syntax:

1 STR User Name

where event memory 1 stores string data and displays it under the label User Name.

A sufficient amount of event memory space for storage of the strings should be allocated. This includes one 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.

Script Conventions

The following are 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 the documentation 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 3-1 lists the characters used in the script syntax:

Table 3-1. Script Syntax Characters

Character	Meaning	Example
( )	Encloses arguments in an instruction	load (ch.ONE,ch.TWO)
,	Optional character that separates the arguments of an instruction	tchars (ch.ONE, 'F')
.	Required syntax character that separates an argument type and argument value	r.1 — argument type is a register, register number is 1
*	Preceding a type, signifies that the value is the number of a script register containing the user space address. A type without an asterisk signifies that the value is an address.	*ch.1 — character string at address in register 1 ch.1024 — character string at address 1024

Destination and Source Arguments

The address modes are represented in Table 3-2.

Table 3-2. Destination and Source Addresses

Address	Modes
type.dst	All types except immediate and time
type.src	All types
ctype.dst	Only types char and *char
ctype.src	Only types char and *char

Arguments to Script Instructions

Acceptable abbreviations for argument types are shown in the Table 3-3. The format of an argument is:

type.#

Type signifies one of the arguments types listed in Table 3-3; # is replaced by a numerical value or a #define symbol. Refer to Chapter 2, "Development Guidelines" for additional information about define statements.

Address Modes

The data types are summarized in Table 3-3. 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. *type.dest* or *type.src* 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 3-3. Argument Data Type

Argument Type	Field Width (Bytes)	Meaning	Example
immed†	-	Actual value, for example, a number, string, or string address	im.4, imm.”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	ev.1
*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
X	-	Data passed via the exec instruction. A value following (.) is ignored. See **exec** in Appendix A, "Summary of Script Instructions"	X.0
Chan	4	The channel number on which the script is running. A value following (.) is ignored.	Ch.0

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

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.

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 T/R, T1, and signal processing (SP) 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, "Development Guidelines"), 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 3-4. The default for the inflection is m for medial.

Table 3-4. 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*])**

  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.

  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 standard 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 which support English only.

  The **tnum** instruction uses the same arguments for inflection as the **tchars** instruction (see Table 3-4).

  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 standard 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, im.4)
tnum(int.FOUR,'f')
```

- **talk("*phrase_ name*"[,"*phrase_name*"]...)**

  The **talk** instruction uses one or more *“phrase_name”* arguments. Each argument is a group of words enclosed in quotation marks. When the **talk** instruction is executed, each *“phrase_name”* argument is found in the listfile designated by the previous **tfile** instruction and is queued for playing.

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

```
talk("Hello this is the CONVERSANT VIS")
talk("Please enter your ID")
```

  The *“phrase_name”* arguments must match a phrase in the ***application_name*.pl** file (see the **tfile** instruction). For example, the ***application_name*.pl** would also contain these matching phrases.

```
Hello this is the CONVERSANT VIS
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, CONVERSANT Voice Information System")
talk("h, c v 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 v s")
talk("Hello Voice Information System")
```

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 C 1")
talk("H C 1 V")
```

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 T/R and the SP 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**, **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 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 3-5. If **tflush** is used without any arguments, the default value of 0 is used for all arguments.

Table 3-5. 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 (refer to "Flow Control Instructions" in this chapter 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',im.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()**

  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 AT&T 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*)**
  background(*type.phrase*,*type.src*)

  NOTE:
  A time division multiplexor (TDM) bus and an SP circuit card must be installed in the system for the **background** instruction to function properly. Refer to the *Intuity CONVERSANT VIS V5.0 Hardware Installation* 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.

  If this phrase is not playing already in the VIS, 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.

  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 SP 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.

NOTE:
On a T/R channel that is not using the TDM bus to play speech (for example, a channel set to “talk,” not “tdm”), foreground speech interrupts background speech. If the TDM bus is used, background speech is heard continuously.

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",im.ADD)
background(im.201,im.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.

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.

- **getdig*(type.type,ctype.dst,type.number)***

  The **getdig** instruction receives information entered by a calling party using touch tones or speech input.

  Type 0 specifies no speech recognition. In this situation, the caller must use touch-tone input. However, touch-tone input is accepted for any type.

  The *ctype.dst* argument specifies the script address (the offset) where the entered touch-tone digits will be stored.

The *number* argument specifies the maximum number of touch-tone digits to be received. Received touch tones are stored as a null-terminated character string in a buffer specified by the destination argument.

The **getdig** instruction has a 10 second default wait time for touch-tone input. If the caller does not enter a touch tone or speak a word within the allotted time period, **getdig** returns the number of digits received before the time-out occurred. Use the **tttime** command to specify a desired wait time.

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

When this instruction terminates, a return code is placed in r.0. The following list shows the return values for touch-tone input, where *N* represents the number of touch tones received:

N > 0	If the *number* argument is greater than 0 (fewer than the expected number of touch tones were received), an interdigit time-out occurred.
N = 0	An initial time-out occurred.
N < 0	A system error occurred.

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

```
getdig(0,ch.ANS,10) /*Get up to 10 touch tones and */
                    /*store at address ANS*/
```

For information about the **getdig** instruction in relation to speech recognition, refer to “Feature Related Instructions” later in this chapter.

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

  The **tttime** instruction allows a script to set the time-out values for 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 **getdig** 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 **getdig** instruction. If the *firstdig* time is exceeded, r.0 is set to 0 and the **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, and the **getdig** 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(im.10,im.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 **getdig** 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 **getdig** and **tttime** instructions. For example, after requesting five digits with a **getdig** 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 delimiters that are in effect when a **getdig** instruction is executed.

The values for the **ttdelim** arguments are shown in Table 3-6:

Table 3-6. 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 # * 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 **getdig** 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 **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, **getdig** would require a minimum of 7 digits to accommodate the two-digit delimiter for *erase-all*.

Based on the previous arguments for the sample **ttdelim** instruction shown earlier, the **getdig** instruction would have the results given by the examples in Table 3-7.

Table 3-7. getdig 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 *im."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(im."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(im.0)
  ```

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

Sample Script Using Data Gathering Instructions

In this example, the script instructs the VIS 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 VIS 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.")
getdig(0,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, refer to Appendix A, "Summary of 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 UNIX system form to “tm” structure form. The *type.src* argument should contain a number representing the UNIX 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 *UNIX SVR4.2 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 UNIX 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 (im.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")
    getdig(0,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,im.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,im.0)
    talk("Enter your password.")

validate:
    getdig(0,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,im.1)
    jmp(r.1 == im.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(im."abc",im."abx")
strcmp(im."abx",im."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 im."Welcome" is copied to the destination ch.COPY.

  ```
  strcpy(ch.COPY,ch.ORIGINAL)
  strcpy(ch.COPY,im."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< im.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. Refer to Appendix A, "Summary of 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 is detected for a T1 or Primary Rate Interface (PRI) channel.

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

  The **exec** instruction allows a script to execute another script.

  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 script. 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 used 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. Refer to Appendix A, "Summary of Script Instructions" for more information on the **exec** instruction.

- **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 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 is used to overlay the actual data of the parent script.

- **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
<	Less than
<=	Less than or equal to
>	Greater than
<=	Greater than or equal to

- **<*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.

- **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.

  - **dbase**
  - **phreserve**
  - **tic**

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

  The **quit** instruction terminates the script voluntarily. An optional parameter is passed to a DIP specified in a **dipterm** instruction.

- **rts()**

  The **rts** instruction is the mechanism for returning from a subroutine call. The saved values for r.1, r.2, and r.3 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.

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

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

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,im.0) /*initialize loop counter to 0*/

start:
talk("To speak to an operator, enter 1")
talk("To hear your account balance, enter 2")
getdig(0,ch.DECIDE,1)
case(ch.DECIDE,im.'1',OPERATOR,continue)
case(ch.DECIDE,im.'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(im.20) /*maximum seconds to wait for host
confirmation*/
--
--
(query host)
--
--
talk("Your account balance is")
tnum(int.FIVE,'f')
rts()

continue:
ibrl(im.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 **getdig** 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 im.COUNTER (which was set at 2 at the beginning of the script) with the value in r.1. If r.1 is less than imm.COUNTER, the script returns to the start code. If r.1 is equal to imm.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. Refer to "Voice Output Instructions" in this chapter 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 3-8:

Table 3-8. Coding Designations

Coding Style	Description
ADPCM32	Adaptive differential pulse code modulation at 32 Kbps
ADPCM16	ADPCM at 16 Kbps
SBC16	Sub band coding at 16 Kbps
SBC24	Sub band coding at 24 Kbps
PCM64	Pulse code modulation at 64 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*)**

  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 to play a short beep just before voice coding starts.

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 (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. Refer to "Voice Output Instructions" for details on the **tflush** instruction.

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")
getdig(0,ch.TALK,2) /*get talkfile id*/
atoi (int.TFILE,ch.TALK) /*convert TT number to integer value*/
phreserve(im.200,int.TFILE,im.100,im.ADPCM32) /*create phrase 200*/
vctime(im.5,im.10) /*initial timeout 5 seconds*/
     /*interword timeout 10 seconds*/
vc('b',im.100,im.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(im.200,int.200) /*remove phrase just created*/
quit()
```

Network Interface Instructions

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

tic

- **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*)**

NOTE:
The **tic** instruction is also used by optional features available with the Intuity CONVERSANT VIS. Refer to "Feature-Related Instructions" later in this chapter for additional information.

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 r.0 and r.1 to return a result. This result may differ according to whether the script is using a T/R, T1, or PRI channel. Where such variations exist, they are noted below.

- *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 any call progress tone, then resume the script
- *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

For more information about the **tic** instruction, refer to Appendix A, "Summary of Script Instructions" Refer to “Full CCA Instructions” later in this chapter for information on the **tic** instruction when Full CCA is used and “PRI Instructions” for information on the **tic** instruction when PRI is used

The following 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, imm."9999")
tic('O', ch.NUMBER)
jmp(r.0 == imm.-1, end) /* hardware failure */
jmp(r.0 == imm.-2, end) /* timeout, no response */
jmp(r.0 == imm.'B', end) /* busy */
jmp(r.0 == imm.'R', ok) /* ring */
end:
  quit()
ok:
  tic('h')
  rts()
```

Sample Scripts 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 the time set by the value of im.2. The **tic** then dials out on the current channel using the phone number stored in ch.PHONE_NUM. The script sleeps for the time specified in im.3, then quits.

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

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

In the following example, the script asks the caller to enter a 5-digit account number, press * (star) to speak to an attendant, or press # to end the transaction. The script then defines the touch tones * and # using the **ttdelim** instruction. The script waits for seven touch tones and stores them in ch.INPUT. The script compares the contents *ch.0 with im.'*' and im.'#'. If *ch.0 is equal to imm.'*', the script goes to the attendant code. If *ch.0 is equal to im.'#', the script goes to the BYE subroutine.

```
#define INPUT 0
#define PHONE_NUM 5551234
.
.
.

start:
   talk("to check your balance, enter your 5-digit account
number")
   talk("to speak to an attendant, press *")
   talk("to terminate your call, press #")
   ttdelim(-1,-1,'*','#')
   getdig(0,ch.INPUT,7)
   decr(r.0,im.1)
   incr(r.0,im.INPUT)
   jmp(*ch.0==im.'*', attendant)
   jmp(*ch.0==im.'#', BYE)
   jmp(r.0==im.5, chk_acct)
   goto(invalid_entry)
```

```
attendant:
   talk("when you are finished speaking with the attendant")
   talk("press one to return to this service")
   tic('a') /*take line off hook*/
   sleep(im.2) /*wait for dial tone*/
   tic('d',ch.PHONE_NUM) /*dial attendant*/
   tttime(180,0) /*script will sleep up to 3 minutes*/
   getdig(0,ch.SLEEP,1)
   jmp(ch.SLEEP==im.1, start)
   jmp(ch.SLEEP==im.'#', BYE)
   .
   .
   .
```

Feature-Related Instructions

The following information details new script instructions and script instructions that have been modified to support Intuity CONVERSANT VIS features.

Full CCA Instructions

The following are instructions used by the Full CCA feature:

- **setcca*(type.mode,type.nrings,type.ansdet*)**

  The **setcca** script instruction allows application developers to set Full CCA parameters at the script level. The *type.mode* argument specifies the type of call classification analysis to be used, either intelligent or full. The *type.nrings* specifies the number of rings to wait for answer (1–10 rings). The *type.ansdet* specifies whether to turn on answer detection. Valid values are 0 for no and 1 for yes. The default is 1 [yes for T/R and Line-Side T1 (LST1) lines, no for T1 and PRI]. By default, answer detection is turned on (value 1) for T/R and LST1 lines and off (value 0) for T1 and PRI lines because T/R and LST1 lines do not have answer supervision while T1 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'** or **tic'w'** instructions.

- **tic**

  When an outbound application uses Full CCA, be aware that if Full CCA determines that the 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.

Refer to Appendix A, "Summary of Script Instructions" for a list the possible return values for the **tic('D')**, and ***tic('O')*** instructions when Full CCA is turned on via the **setcca** instruction. The set of possible return values depends on the type of channel (T1, PRI, T/R, or LST1).

Appendix A also lists the ISDN Cause Values returned in r.1 for a given call disposition returned in r.0 of the tic instruction when an outdial is used and the special information tones (SITs) that are returned to the script in r.1

Sample Script Using setcca and tic for Full CCA

The following example is an excerpt from a script showing how the **setcca** and **tic** instructions can be used in an application.

```
setcca(im.1,im.10,im.-1)
nextcall:
dbase( .... )     /* get number to dial from DIP */

tic('O', r.3)    /* call number in register 3 */

jmp(r.0 == im.'N', noAns)     /* no answer after 10 rings */
jmp(r.0 == im.'B', busy)
jmp(r.0 == im.'F', retry)
jmp(r.0 == im.'A', answer)
jmp(r.0 == im.'s', SIT)
jmp(r.0 == im.-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 == im.'R', retry)
jmp(r.1 == im.'r', retry)
jmp(r.1 == im.'K', retry)
jmp(r.1 == im.'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)
```

WholeWord Script Instructions

The following are instructions used by the WholeWord speech recognition feature:

- **getdig*(type, ctype.dst, number, ctype.mode)***

  The generic **getdig** instruction has some new options for WholeWord speech recognition arguments. **Getdig** receives touch-tone or speech information entered by a caller.

  The first argument, *type*, specifies whether touch-tone or speech input is expected from the caller. Type 0 specifies 12-key telephone touch-tone input. A non-zero value for *type* specifies speech input. The **getdig** instruction requires the recognition type used for a particular grammar. The choices available for *type* in this instruction can be found in the file **/att/include/sr_grammar.h**. The grammars and their specifying values are listed in the *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228.

  NOTE:
  For packages which support connected-digit recognition: For US English, you may use US_0_9o to recognize any variable-length string of 1–24 digits. If the string length is known in advance, however, superior recognition performance can be obtained by using one of the grammars with a fixed string length.

  If the *type* argument is 0, the *number* argument specifies the maximum number of touch-tone digits to be received. The maximum value is 128. Received touch tones are stored as a null terminated character string in a buffer specified by the destination argument, *dst*.

  If the *type* argument is other than 0, the number argument specifies the maximum string length for speech input. Received speech input is stored as a null terminated character string in a buffer specified by the destination argument. The characters are defined by the vocabulary provided. Possible characters are listed in *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228.

  NOTE:
  For packages which support connected-digit recognition: This maximum value of the string length for speech input is 24.

  The fourth argument, *ctype.mode*, indicates to the script whether the response is touch-tone or voice. If the response is touch-tone, -1 is stored in *ctype.mode*. If the response is voice, then the number (non-negative) of the SP card that recognizes the voice is stored in *ctype.mode* to be used later by a DIP.

  When the **getdig** instruction terminates, a return code is placed in r.0. Appendix A, "Summary of Script Instructions" lists the return codes for speech input.

- **sp_alloc*(type.onoff, type.resource)***

  The **sp_alloc** script instruction is used to allocate or deallocate speech recognition on the SP circuit card.

  NOTE:
  For CONVERSANT Version 5.0, only the WholeWord and FlexWord SP resources should be used with the **sp_alloc()** instruction.

  NOTE:
  The **sp_alloc** instruction has replaced the **sr_alloc** instruction . Since **sr_alloc** is being replaced, the following warning message appears on the screen when the **sr_alloc** script instruction is used:

  ```
  WARNING: sr_alloc is obsolescent. Use sp_alloc.
  ```

  Although the **sr_alloc** instruction will compile, application developers are encouraged to use the **sp_alloc** instruction.

  The **sp_alloc** instruction may be used by a script to allocate the speech recognition resource on the SP 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 SP resource is not available when an instruction that requires it is executed, the instruction will fail. By using **sp_alloc**, the script may test for the availability of a particular SP resource. If the resource is available, it will be allocated to the script until the script terminates or until the script deallocates the SP resource using **sp_alloc**.

  sp_alloc may be used to allocate an SP resource for a period longer than the script is actually recognizing speech. Avoid overloading the system's SP facilities 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 SP resource

  0 Deallocate the SP resource

The *type.resource* argument is used to tell the s**p_alloc()** which SP resource or combination of SP resources to allocate or deallocate. Each SP resource has a unique value. The values for each resource and examples of how resources can be added are listed in Appendix A, "Summary of Script Instructions"

- **sr_talkoff*(im.1 | im.0)***

  The **sr_talkoff** instruction is used to enable or disable speech recognition during the prompt.

  If speech recognition during the prompt is enabled by using **sr_talkoff(im.1)**, the **getdig** instruction begins playing any phrases in its queue and simultaneously turns on the recognizer. If the recognition during prompt is disabled by using **sr_talkoff(im.0)**, first, the **getdig** plays any phrases in its queue, and then it turns on the recognizer.

  If recognition during prompt is enabled, the call can be received through a telephony interface card (T1 or T/R) that is connected to the TDM bus with the ‘‘tdm’’ option set. Enabling **sr_talkoff** requires the use of the SP circuit card to play the prompts. This is already set for T1. However, for T/R cards set to ‘‘talk,’’ the system detects a ‘‘recognition during prompt’’ request in the script and automatically uses an SPcircuit card to play the prompts. Settings and their results are as follows:

  “talk” set + sr_talkoff “off” = the system plays all prompts with the T/R circuit card

  “talk” set + sr_talkoff “on” = the system plays all prompts with the SP circuit card

  Playing prompts uses resources on the SP circuit card. Application developers who find that these SP resources are strained may want to consider configuring their T/R circuit cards with ‘‘talk’’ and designing the application to use as few SP resources as possible.

  Script register 0 (r.0) is set to the following values to indicate the status of **sr_talkoff** execution:

  0 Success

  -1 Failure

  -2 System resource not available

FlexWord Script Instructions

The following are instructions used by the FlexWord speech recognition feature:

- **getdig(*type, ctype.dst, number, ctype.mode)***

  The generic **getdig** instruction has some new options for FlexWord speech recognition arguments. **Getdig** receives touch-tone or speech information entered by a caller.

  The first argument, *type*, specifies whether touch-tone or speech input is expected from the caller. Type 0 specifies 12-key telephone touch-tone input. A non-zero value for *type* specifies speech input. The **getdig** instruction requires the recognition type used for a particular wordlist. The choices available for *type* in this instruction can be found in the **vs/data/sr_file**. The **sr_file** contains a line for every wordlist installed on your system. If WholeWord is installed, this **sr_file** also lists all of the grammars on the system. Information in this file maps wordlist names into the symbols that you need to use with this instruction. Therefore, for a specific wordlist, look to the fourth field and use this “defined symbol,” which correlates with the wordlist on the same line.

  If the *type* argument is 0, the *number* argument specifies the maximum number of touch-tone digits to be received. The maximum value is 128. Received touch tones are stored as a null terminated character string in a buffer specified by the destination argument, *dst*.

  If the type argument is other than 0, the number argument specifies the maximum string length for speech input. Received speech input is stored as a null terminated character string in a buffer specified by the destination argument. The characters are defined by the words in the wordlist.

  The fourth argument, *ctype.mode*, indicates to the script whether the response is touch-tone or voice. If the response is touch-tone, -1 is stored in *ctype.mode*. If the response is voice, then the number (non-negative) of the SP card that recognizes the voice is stored in *ctype.mode* to be used later by a DIP.

  When the **getdig** instruction terminates, a return code is placed in r.0. The return values for touch-tone input and speech input are shown in Appendix A, "Summary of Script Instructions"

- **sp_alloc*(type.onoff, type.resource [,type.mode])***

  The **sp_alloc** instruction is used to allocate or deallocate speech recognition on the SP circuit card.

  NOTE:
  For CONVERSANT Version 5.0, only the WholeWord and FlexWord SP resources should be used with the **sp_alloc** instruction.

  NOTE:
  The **sp_alloc** instruction has replaced the **sr_alloc** instruction. Since **sr_alloc** is being replaced, the following warning message appears on the screen when the **sr_alloc** instruction is used:

  ```
  WARNING: sr_alloc is obsolescent. Use sp_alloc.
  ```

  Although the **sr_alloc** instruction will compile, application developers are encouraged to use the **sp_alloc** instruction.

  The **sp_alloc** instruction may be used by a script to allocate the speech recognition resource on the SP 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 SP resource is not available when an instruction that requires it is executed, the instruction will fail. By using **sp_alloc**, the script may test for the availability of a particular SP resource. If the resource is available, it will be allocated to the script until the script terminates or until the script deallocates the SP resource using **sp_alloc**.

  sp_alloc may be used to allocate an SP resource for a period longer than the script is actually recognizing speech. Avoid overloading the system's SP facilities 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 SP resource
0	Deallocate the SP resource

The *type.resource* argument is used to tell the **sp_alloc** which SP resource or combination of SP resources to allocate or deallocate. Each SP resource has a unique value. The values for each resource and examples of how resource values can be added are listed in Appendix A, "Summary of Script Instructions"

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 **Defines.h**) — Allocate resources immediately
- IRD_BLOCKFOREVER (defined in **Defines.h**) — Wait until resource becomes available before continuing
- <*n*> — Wait <*n*> hundresths of a second for resource to become available fore continuing, where *n* is a positive integer

Text-to-Speech Script Instructions

The **say** instruction is used by the Text-to-Speech (TTS) feature to direct the VIS to speak ASCII text stored in a buffer. The format of the **say** instruction is:

say(*ctype.src*)

where *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):

Table 3-9. 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.

PRI Script Instructions

The following are instructions used by the PRI feature:

- **tic**

  The supported tic instruction options are listed in Table 3-10. These options are used in the same manner for PRI as for T1.

Table 3-10. tic Options Supported for the PRI

Option	Function
a	Answer an incoming call
h	Disconnect (hangup) a call
o	Originate a call
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 3-11.

Table 3-11. 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 PRI implementation of the **tic('O')** (Originate) instruction provides additional return code information over the T1 and T/R 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 A-11 in Appendix A, "Summary of 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 calling party number (CPN) from the network before starting the script. If the switch is provisioned to allow you to select the type of CPN on incoming calls, the following types of CPN 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,imm."6145555121")
       jmp (r.0 == imm.0, gotdnis)
       setattr (ATTR_ANI)
     gotdnis:
```

The values in Table 3-12 (defined in **tas_defs.h**) can be used with the **setattr** instruction to specify the type of CPN being requested.

Table 3-12. 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”     /* VIS 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 != imm.’6’, WrongAreaCode )
     jmp (ch.NUMBER_TO_BE_CHECKED+1 != imm.’1’, WrongAreaCode )
     jmp (ch.NUMBER_TO_BE_CHECKED+2 != imm.’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 VIS, 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 3-1 depicts a call placed to “1234” that has been redirected to “5678.” In this example, the VIS provides the telephone numbers shown in Table 3-13 to the script.

Table 3-13. Telephone Numbers Provided the Script for Redirected Numbers

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

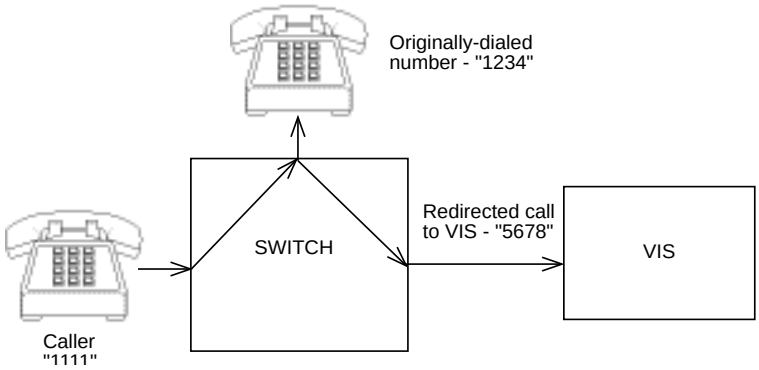


Figure 3-1. Retrieving the Redirection Number - Example

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 stored in the IE_SERVICE register as an integer. Valid service types are listed in Appendix A, "Summary of 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, imm.”6145551212”)
  tic (‘o’,ch.CALLED)
  ```

  NOTE:
  If the **setstring** command fails, r.0 is set to a negative number. Note that the setstring command faild 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 for outbound PRI calls by setting the SERVICE_TYPE parameter. Valid parameters and their values are listed in Appendix A, "Summary of 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.

  This instruction returns in r.0 the following positive integer values from the **/att/include/irDefines.h** header file:

Table 3-14. chantype Return Values

Value	Channel Type
IRD_TR	Tip/Ring
IRD_T1	T1 E&M protocol
IRD_PRI	IDSN PRI protocol
IRD_LST1_DEF	Line Side T1 for DEFINITY
IRD_LST!_GAL	Line Side T1 for Galaxy
IRD_LST1_ASAI	Line Side T1 for DEFINITY with ASAI
IRD_ASAI	Tip/Ring with ASAI
IRD_VIRT_CHAN	Virtual channel

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 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").

  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 (imm.45, imm.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").

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 scirpt uses the event instruction. See the event instruction in this chapter or refer to Appendix A, "Summary of Script Instructions"

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

The **trace** script instruction works with the trace 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, imm.25001 and int.F_TEMP are traced.

```
trace (imm.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.

Refer to the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230, 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 (refer to *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230, 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 **getdig**:

```
getdig( SYNONLY, ch.YN, 1 )  /* Request Yes/No response */

incr(ev.1, im.1 )            /* Record event 1 */

load(ev.2, int.NAME )        /* 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.

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 Chapter 2, "Development Guidelines" and the *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230.

Script Instructions and Wait Conditions

The TSM program is responsible for running scripts that are produced with an editor or Script Builder 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**, **tchar**, **tnum**, 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()
nwitime()
phremove()
phreserve()
quit()
setalk()
sleep()
sr_talkoff()
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 **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. Refer to 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 **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 delimiters that allow **getdig** to accept variable length digit strings without waiting for a timeout. Two timeouts affect the **getdig** instruction: an initial timeout and an interdigit timeout. Both of these timeouts are five seconds by default. Change the default timeout using with the **tttime** instruction.
- The **phreserve** instruction is used to reserve space in a talkfile for a phrase. The script waits for VROP to complete the request. The timeout period for **phreserve** is 45 seconds by default. Change the timeout value with the **nwitime** instruction.
- 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 **talkresume** instruction is used to play speech that is *remembered* for the channel by VROP (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. VROP 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 TSM's event queue is empty. These events may not occur under a heavy system load, preventing TSM from continuing script execution. As a result, your script could be suspended indefinitely. 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**, **sp_alloc**, **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"). 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 the description of **scrinst** earlier in this chapter 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

Refer to *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230, for more information on the **trace** command.

Check the Status of talk Instructions

One or more **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 VIS 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 **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 phrase buffer becomes full
- 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 **talk** instructions. The **tflush** instruction, like other speech-flushing instructions, causes the phrases queued by the **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 < im.0,play_fail)
        getdig(0, 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 **talk** instructions have executed, not after every **talk** instruction. Each time **tflush** is executed, two interprocess messages are sent: TSM sends a message to VROP which causes the phrases to be played and VROP returns a message to TSM which contains the status. These processes can delay play of a script if **tflush** is used too often.

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 **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. Refer to the description of the **setttfl** instruction earlier in this chapter.

Data Interface Process

4

What's in This Chapter

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 TSM script instructions.

This chapter assumes the following:

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

NOTE:
The information provided in this chapter is provided for backward compatibility and support of DIPs created in releases prior to Intuity CONVERSANT Voice Information System (VIS) V5.0. All new DIPs should be written in terms of the IRAPI. Refer to the *Intuity CONVERSANT VIS V5.0 IRAPI Programming Guide*, 585-310-226, 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 TSM 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 UNIX 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 4-1 shows typical interaction between a DIP and other processes in the voice system software.

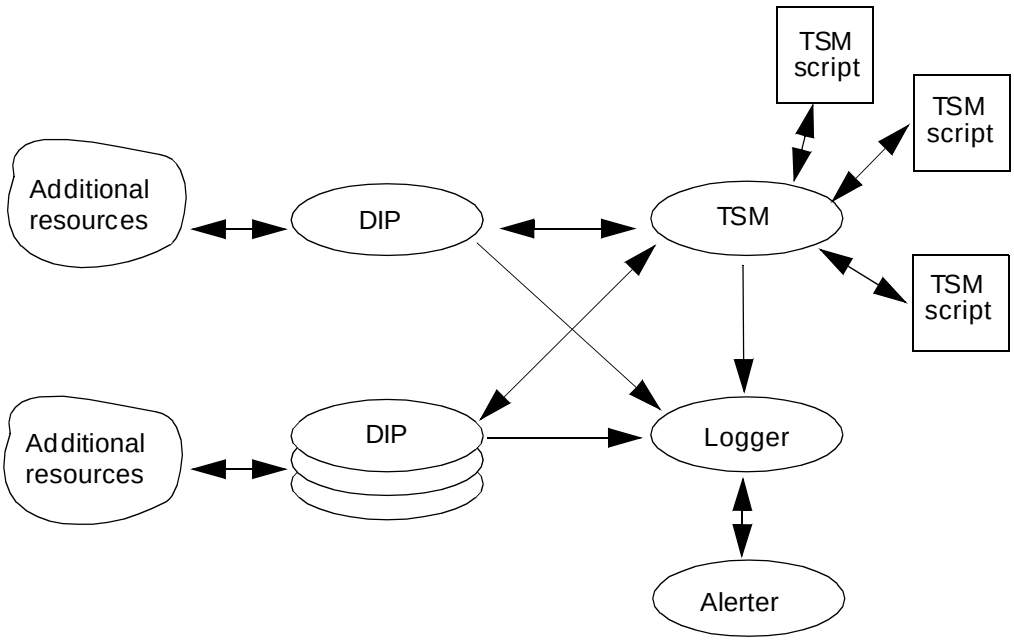


Figure 4-1. Data Interface Process Architecture

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 4-1 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 UNIX 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" section of this chapter.

For More Information About Message Queues...

For additional information about UNIX System Message Queues, refer to the *UNIX SVR4.2 Programming with UNIX System Calls*. For additional information about UNIX programming features, refer to the *UNIX SVR4.2 Programming in Standard C*.

The following book may be purchased at a commercial bookstore:

Advanced UNIX Programming
Rochkind
Prentice-Hall ISBN 0-13-011800-1

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 they are run each time on the voice system.

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 UNIX 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 VIS. This usually frees some slots so that you can post your process. Refer to the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230, 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. Write the C-code to define the data to be passed between the DIP and the TSM script.
2. Add the C-code that initializes the DIP to the voice system.
3. Add the C-code to send and receive messages.
4. Add the C-code to implement the application-specific processing.
5. Define and add logger errors that this DIP will send in the voice system.
6. Add error reporting C-code to notify the logger of errors.
7. Add C-code trace messages.
8. Compile and execute the DIP.

Steps 1–3, and 7–8 are covered in detail in this chapter. Steps 5 and 6 are covered in Chapter 5, "Adding and Modifying System Messages". Step 4 is the responsibility of the application developer.

The examples provided here are used in the template for writing a DynaDIP. See the "Sample DIP" in Chapter 7, "Application Example".

Step 1: Defining Data to be Passed Between DIP and 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 4-2.

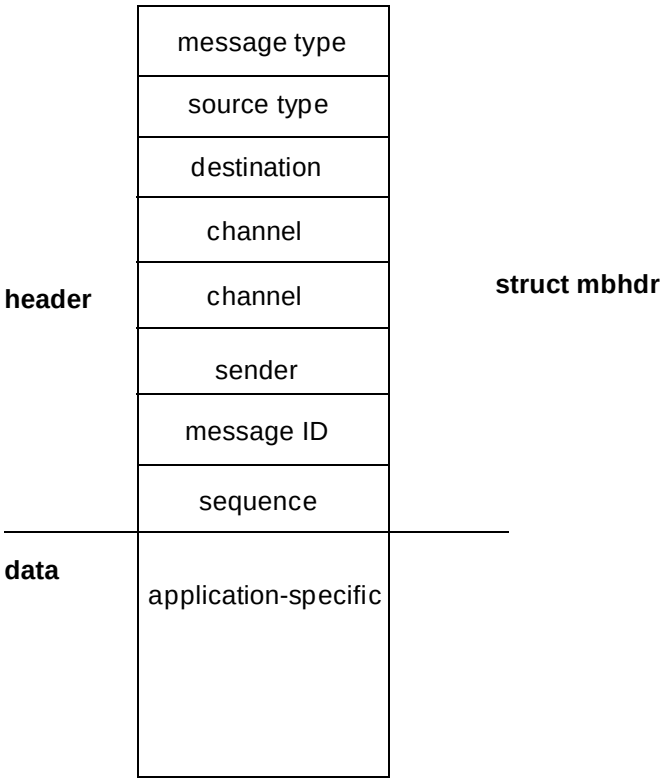


Figure 4-2. Voice System Message Components

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 4-3 displays the header structure for these IPC messages.

```
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 */
};
```

Figure 4-3. Message Header Structure

NOTE:
As you read the following field descriptions, refer to the "Sample DIP" provided in Chapter 7, "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 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.
- 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.
- 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.
- 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, which allows the handling of messages with data of all shapes, sizes, and meanings. Message ids are usually found in the DIP's **.h** file (***<dip_name>*.h**). Without this message id, the TSM script or the DIP are unaware of kind of data received.

  For an application to identify each different message to be sent or received, select a unique positive number. In the **mesgrcv** example in Appendix B, "Voice System 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 is 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 Figure 4-4. This figure uses an example from a stock application.

```
/* Define all message structures that can be received. */
struct stockInfo {
       struct mbhdr  hd;
       int  stockId;
};
struct callerInfo {
       struct mbhdr  hd;
       char   callerName [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 4-4. Message Structure Union Example

Figure 4-4 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 4-5.

```
/* universal structure for passing a message */
struct ms_univ {
        struct mbhdr   hd;
        long      arg[4];
};
```

Figure 4-5. Standard Message Structure

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

Step 2: Initializing

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 4-6. A negative value is returned if an error occurs.

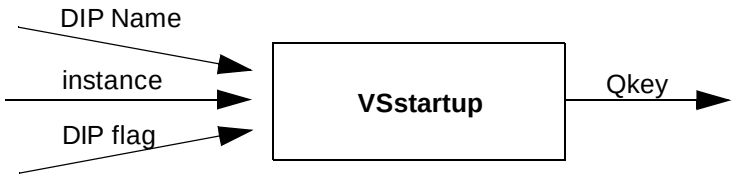


Figure 4-6. VSstartup Input and Output

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

Figure 4-7 displays the **VSstartup** synopsis in C-code for the dynamic DIP.

```
#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? */
```

Figure 4-7. VSstartup Synopsis

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 on **VSstartup**, refer to Appendix B, "Voice System 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 in Appendix B, "Voice System 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 4-8 and Figure 4-9.

Figure 4-8. VStoqkey Input and Output

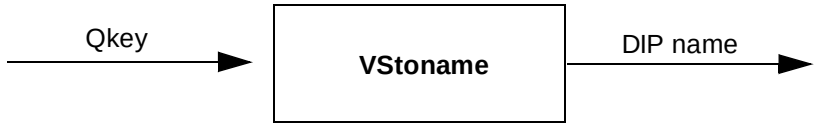


Figure 4-9. VStoname Input and Output

Figure 4-10 and Figure 4-11 display the **VStoqkey** and **VStoname** synopsis in C-code for the dynamic DIP.

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

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

Figure 4-10. VStoqkey Synopsis

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

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

Figure 4-11. VStoname Synopsis

For additional information on **VStoqkey** and **VStoname**, refer to Appendix B, "Voice System 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 4-13 displays the **VSerror** synopsis written in C-code for the DIP.

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

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

Figure 4-12. VSerror Synopsis

For additional information on **VSerror**, refer to Appendix B, "Voice System C-Library Functions"

Hardcoded DIPs

startup

The **startup** function is called once to post a hardcoded DIP to the BB. As shown in Figure 4-13, startup takes the Qkey and slot_offset as arguments.

```
#include "spp.h"

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

Figure 4-13. startup Synopsis

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 on **startup**, refer to Appendix B, "Voice System 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: Sending and Receiving 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 4-14, **mesgsnd** takes a pointer to the message *msgp* of size *msgsz* bytes and sends it to the message queue identified by the Qkey *mdest*.

```
#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 */
```

Figure 4-14. mesgsnd Synopsis

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 UNIX system call **msgsnd**, (see the *UNIX SVR4.2 Operating System API Reference)* 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 on **mesgsnd**, refer to Appendix B, "Voice System 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 4-15. The **mesgrcv** function creates the message queue if necessary.

```
#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 */
```

Figure 4-15. mesgrcv Synopsis

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 Chapter 7, "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 arrived if none are 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 UNIX 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 on **mesgrcv**, refer to the Appendix B, "Voice System C-Library Functions". For additional information on the UNIX system call **msgrcv**, refer to the *UNIX SVR4.2 Operating System API Reference*.

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" in Chapter 7, "Application Example".

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" in Chapter 3, "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 to 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**, describe how a TSM script accomplishes these tasks. For additional information on these instructions, refer to Appendix A, "Summary of Script Instructions".

NOTE:
A Script Builder external function can be written to send and/or receive messages from DIPs. Refer to Chapter 12, “Using Advanced Features,” of *Intuity CONVERSANT VIS Script Builder*, 585-310-727, for tips on writing external functions.

- **dbase**

  Both the sending and receiving of data is done through the **dbase** instruction. **Dbase** 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.

  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**

  dipterm instructs TSM to send a message to the specified DIPs 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** message is defined as the C-structure struct **ms_univ** (see **mesg.h**). Figure 4-16 and Figure 4-17 show the fields of the message and their values as set by TSM.

```
/* message structure for dipterm message */
struct ms_univ {
       struct mbhdr  hd;
       long          arg[4];
};
```

Figure 4-16. dipterm Synopsis

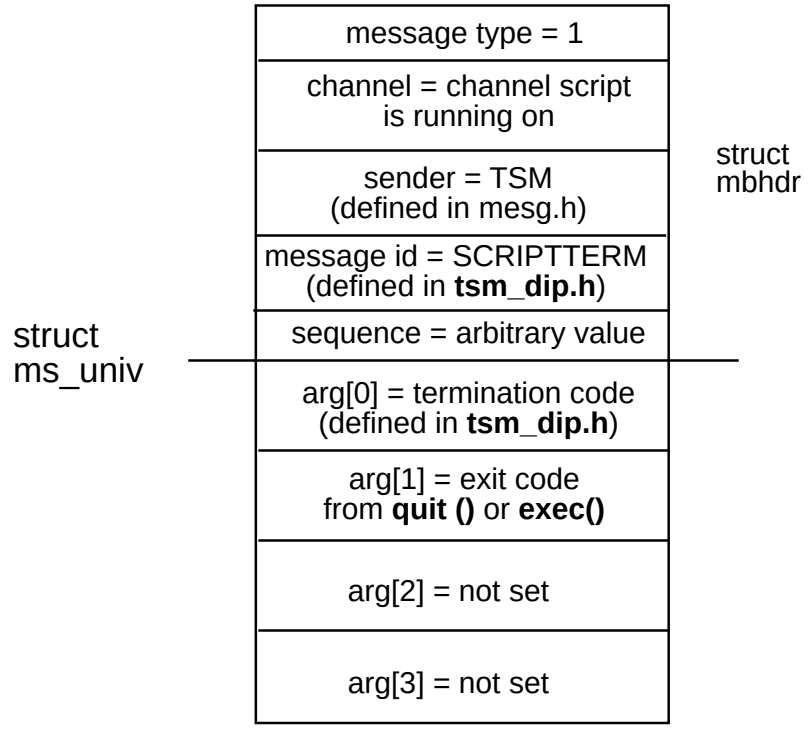


Figure 4-17. dipterm Message Structure

The *arg[0]*, as shown in Figure 4-17, 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**

  The dipname function takes a DIP number and converts it to the corresponding DIP name, as shown below:

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

  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**

  dipnum converts a DIP name to its corresponding DIP number, as shown below.

  dipnum*(type.dst,ctype.src)*

  dipnum returns a negative value if an error occurs during translation.

At this point, you have completed the necessary steps to send and receive message to the DIP from a TSM script. Next, refer to Chapter 5, "Adding and Modifying System Messages" for complete information on adding, deleting, and/or modifying logger/alerter messages for your DIP. Then, return here to complete Steps 7 and 8.

Step 7: Tracing DIPs

DIPs can be traced by embedding debug information in the DIP and displaying it via the **trace** command. This feature is enabled by **VSstartup** or **startup**. The debug information should be strategically placed in the DIP code, then 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 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 *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230, 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 4-18.

```
#include "spp.h"

int db_pr(format, arg ...)
char   *format;  /* printf format string */
```

Figure 4-18. db_pr Synopsis

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 instead of printf because it allows a more controlled method of output.

For additional information on **db_pr**, refer to Appendix B, "Voice System C-Library Functions".

db_put

The **db_put** library function applies a single string of an argument and displays it as shown in Figure 4-18.

```
#include "spp.h"

int db_put(string)
char    *string;  /* string to write out */
```

Figure 4-19. db_put Synopsis

db_put trace messages are displayed when tracing is on regardless of what processes are being traced.

For additional information on **db_put**, refer to Appendix B, "Voice System C-Library Functions".

Step 8: Compiling and Executing a 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 \
/vs/lib/libspp.a /vs/lib/liblog.a /vs/lib/libprism.a
```

NOTE:
This information should appear entirely on one line in the file as indicated by the backslashes (\) at end of the lines.

For more information about the C-compiler, refer to the *UNIX SVR4.2 Programming in Standard 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 UNIX 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
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, execute **touch /etc/conf/init.d/CONVERSANT** 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.

- 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. Refer to the *UNIX SVR4.2 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. Refer to the *UNIX SVR4.2 Command Reference* for more information about the **ipcs** command.

- Bulletin board

  Use the **bbs** command to display the posted processes in the BB. See *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230, 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. Refer to the *UNIX SVR4.2 Command Reference* for 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 the information earlier in this chapter on the "Types of DIPs" for details on how to tune your system to handle more processes.

Voice System Hardcoded DIPs

The following tables show the current hardcoded DIPs used in the voice system. The DIP name and number are listed, as are the software packages that interface 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. For example, DIP8 will be available if AUDIX Voice Power is not installed on your system.

Table 4-1. Voice System Hardcoded DIPs

DIP Name	DIP #	Package
agdip3270	DIP0	Host
oraldb	DIP1	Local Database
vmdip	DIP2	AUDIX Voice Power
rptdip	DIP3	AUDIX Voice Power
Spadm DIP	DIP4	Speech Admin
sys_stat	DIP5	CVMS Interface
mtcxmtr	DIP6	CVMS Interface
asaihp	DIP7	ASAI
RADMDIP	DIP8	AUDIX Voice Power
ADMDIP	DIP9	AUDIX Voice Power
xferdip	DIP10	Call Bridge
agdiphelper	DIP11	Host
	DIP12	available
	DIP13	available
	DIP14	available
	DIP15	available
	DIP16	available
	DIP17	available
	DIP18	MTC

Continued on next page

Table 4-1. Voice System Hardcoded DIPs — *Continued*

DIP Name	DIP #	Package
	DIP19	VROP
	DIP20	available
mdp	DIP21	AUDIX Voice Power
	DIP22	available
	DIP23	available
swindcp	DIP24	AUDIX Voice Power
faxcng	DIP25	AUDIX Voice Power
dc.sh	DIP26	Chan 0
		Data Collection
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
ocdip	DIP31	AUDIX Voice Power
	DIP32	Voice
		Workstation
	DIP33	available
	DIP34	available

TTS_DIP

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** is available through Script Builder as the **tts_file** external function.

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**.

Adding and Modifying System Messages

5

What's in This Chapter

This chapter contains:

- Logger environment overview
- Using logger messages in user data interface processes (DIPs)
- Adding and changing system message explain text

Logger Overview

The following section details concepts of the Intuity CONVERSANT logger and procedures needed to add messages to the Intuity CONVERSANT logger.

Logger Road Map

The logger provides facilities which allow UNIX processes to log messages to predefined destinations with desired priorities (refer to Chapter 3, "Configuration Management," of *Intuity CONVERSANT VIS V5.0 Operations,* 585-310-550). Messages are logged from processes coded as C programs, for example, custom DIPs.

Logger messages are typically used to alert administrators or operators of errors encountered as calls are processed. For example, a DIP may report an error to the logger if it got a message from a script that it does not understand. Logger messages can also be used to inform administrators and operators of events completed by the calling process. For example, a DIP may report to the logger that is has successfully started and initialized itself.

There are several steps involved in coding logger messages in custom software. After points in the source code where logger messages have been identified, the exact structure and wording of the message must be determined. This includes determination of 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 might include such things as an error code, channel number, or a reason string, and may be a character string or integer.

System message format text appears in the ***{CLASS}*msg** files found in the **/usr/spool/log/formats** directory. Customer DIPs must use the **APPLmsg** format file.

CAUTION:
No other {CLASS}msg files should be modified or added. Doing so could render the entire logging system inoperable or produce unintelligible messages in the log files.

All messages in the logger system are organized internally by an indexing scheme. The logger header files found in **/usr/spool/log/head** provide the index of the logger message to the C program sending the message. Entries in the logger header files provide the internal index to the DIP. All definitions for customer DIPs must appear in the **logAPPL.h** header file. There must be a definition in **logAPPL.h** for each message used from the **APPLmsg**. These definitions must be sequential starting at 1 and the index in **logAPPL.h** must match the *msgID* in **APPLmsg**. It is important that no other logger header files in the system be modified in any way. However, DIPs are free to use the messages already defined in these files, if they apply to the situation or condition being reported.

The example provided in Chapter 7, "Application Example" shows the following three message definitions in **logAPPL.h**:

```
#define APPL_INITFAIL        logAPPL(1)
#define APPL_MSGSNDERR       logAPPL(2)
#define APPL_UNKNOWNMSG      logAPPL(3)
```

These correlate to three messages found in the **APPLmsg** file, APPL001–APPL003, also shown in Chapter 7, "Application Example".

Message Content/Format Specification

Message content and format is specified in the **APPLmsg** format file found in **/usr/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 may span multiple lines using the '\' character at the end of the line to indicate continuation to the next line.

While tab ('\t') and newline ('\n') characters may be specified in the message text, it is not recommended since output message text formatting is handled by the display messages command.

The standard Intuity CONVERSANT VIS message text appears as follows:

{msgID} {FRU} {EQ} {EQ#} ({MNEMONIC}) {Message Text}

For the case of **APPLmsg**, *{msgID}* is in the form **APPL*NNN*** where *NNN* is a number ranging from 001 to the number of message in the class. *NNN* must match the index for the message it is specifying in **logAPPL.h**. *{msgID}* must occupy 8 spaces; if *{msgID}* is less than 8 spaces it must be right filled with blanks. The **APPLmsg** file has been delivered with placeholders for APPL001 through APPL032. These placeholders may be used by the customer and more may be added if needed. In order to remove message ids from the system, make sure to replace the error message with a place holder or a new error message.

{FRU} is a two character field indicating field replaceable unit. Examples in the Intuity CONVERSANT environment include TR (Tip/Ring), SP (Signal Processing), etc. If it is not necessary to specify a field replaceable unit, use -- as the value in this field.

{EQ} is a two character field indicating the type of resource to which the message applies (CH for channel, CA for card, or -- if not applicable).

{EQ#} is a three digit number to specify the particular card or channel (use --- as the value in this field, if not applicable).

({MNEMONIC}) is the #define symbol used in the **logAPPL.h** file to define this 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. This text may contain many characters of free text optionally embedded with variable text parameters. The message text may span multiple lines by placing the \ (backslash) character at the end of the line.

Message Text 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. The formatting rules for these parameters are the same as those defined for printf(3S) but should be limited to %s and %d. These parameters specifications may appear anywhere within the *{Message Text}* area.

The *{FRU}*, *{EQ}*, and *{EQ#}* fields may also be specified as parameters in the APPLmsg file. If so, the following specifications should be used:

```
{FRU}   %.2s
{EQ}    %.2s
{EQ#}   %.3d
```

It is not expected that most DIPs would make use of the *{FRU}, {EQ}* and *{EQ#}* fields. Therefore, these fields would be usually specified as "--", "--", and "---" (without quotes) respectively.

Messages defined for Intuity CONVERSANT have included an optional <<{NAME},{TYPE}>> field following each parameter specification. This field may be omitted from messages defined in **APPLmsg** since these fields are only specified for future use.

Message Mnemonic Definition

The **logAPPL.h** header file, located in **/usr/spool/log/head,** must be modified to include the new messages specified in the APPLmsg file. New message definitions should be inserted in the file on the line preceding the last #endif statement in the file. See the example of **logAPPL.h** in Chapter 7, "Application Example" for placement of new messages.

In general, **logAPPL.h** entries should look 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}* would be APPL and *{NAME}* would be some 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 and logAPPL(2) correspond to the message defined for APPL002, etc.

NOTE:

Not all messages allocated in **APPLmsg** need to have a corresponding mnemonic defined in **logAPPL.h**. In the delivered **APPLmsg**, many unused message IDs have been allocated. However, all messages which have message text defined for them must have a corresponding mnemonic.

Compiling the Messages in the DIP

The following procedure assumes that the message text has already been added to the **APPLmsg** file and the mnemonic has already been added to **logAPPL.h**:

1. Include the following logger header files in the DIP code.

   ```
   #include "/usr/spool/log/head/log.h"
   #include "/usr/spool/log/head/systemLog.h"
   #include "/usr/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 following format:

   logInit (*program_name*)

   program_name is normally an all upper case representation of the name of the executable (for example, “MYDIP”).

   Refer to Appendix B, "Voice System C-Library Functions" for additional information on **logInit**.

3. Place calls to **logMsg(3x)** within DIP to send specified messages to the logger.

   logMsg (*MNEMONIC,EL_FL,arg1,arg2, . . .*)

   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 display messages output. Use **logCat** to display messages and their corresponding mnemonics. Use **logFmt** to enable and disable the appearance of the message mnemonic. Refer to *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230, for additional information about **logCat**.

   Refer to the Appendix B, "Voice System 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 runtime. In order for this to happen, first include **-D _INSTALLABLE_APPL** in the DIP compilation statement, as shown in Chapter 4, "Data Interface Process". Secondly, include the **Fcn_APPLMSG_START** function in the DIP code:

```
#include <stdio.h>
#include <sys/types.h>

#include "/usr/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 /usr/spool/log/formats
   make -f formats.mk install
   reinitLog

   For more information on **reinitLog**, refer to the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230.

   NOTE:
   If **reinitLog** seems to fail, execute **touch /usr/spool/log/head/***, then try **reinitLog** again.

6. Compile the DIP code by linking the logger library files found in **/vs/lib/liblog.a** and **/vs/lib/libprism.a**. Refer to Chapter 4, "Data Interface Process" 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, “Configuration Management,” of *Intuity CONVERSANT VIS V5.0 Operations,* 585-310-550*.*
8. Test your messages as described in the following sections.

Testing a Single Error Message

To test a single error message, use the following example as a guideline. This example shows how to test the example message APPL001 (APPL_INITFAIL) found in Chapter 7, "Application Example". You should look at the **APPLmsg** file to know what strings are necessary to fill in the error, and the **logAPPL.h** file for the index of the error, [for example, logAPPL(1)].

1. Enter: **cd /usr/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, use **logCat**. Enter **logCat -t3** to see the last 3 error messages. Your message should be among the messages displayed. For example:

```
*  Wed Feb 23 12:30:53 1994 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 three example 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 /usr/spool/log/formats
   logTest < /tmp/appl_errors

3. Check the error log to make sure your messages are correct by entering **logCat -t3**

   System response:

```
*  Wed Feb 23 12:39:56 1994 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 1994 stock_dip logTest.c:418
APPL002  -- -- --- Application DIP 'stock_dip' failed to send
message to script.
*  Wed Feb 23 12:39:58 1994 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 will print an expansion failure error.

For more information about **logTest** and **logCat**, consult the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230.

Adding and Changing Explain Message Text

You may use the text editor or the command line to add and/or change explain message text.

Using the Text Editor

To add a new explain message text using the text editor:

1. Create the explanation text file in the proper directory. It must be in the directory **/gendb/data/explain/*<alphabetic_letter>*** whose letter matches the 1st 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 below.

Name	Alias	12-Character Alias
ADM001	ADM_SYSERR	
ADM002	ADM_MSGERR	
ALERT001	AL_LVL_CHG	
ALERT002	AL_RESET_STATS	AL_RESET_STA
ALERT003	AL_INVALID_THRESHOLD	AL_INVALID_T
ASAI001	A_LINKDOWN	
ASAI002	A_DSCRIPT_TERM	A_DSCRIPT_TE
ASAI003	A_LOGIN_FAIL	

3. Enter **:wq!** to save the information and exit the file.

Using the Command Line

To add or modify an explain message text using the **edExplain** command:

1. Login to the system as root.
2. At the system prompt, type **edExplain *<msg_id>***

   where *<msg_id>* is the messages id (for example, APPL001).

   Refer to the **edExplain** command in the *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-310-230.

Removing Error Messages

To remove error messages, use the following steps:

1. Edit **/usr/spool/log/head/logAPPL.h** file and remove the #define statements for the custom error messages.

   In the example in Chapter 7, "Application Example" the statements that need to be removed are:

   #define APPL_INITFAIL logAPPL(1)
   #define APPL_MSGSNDERR logAPPL(2)
   #define APPL_UNKNOWNMSG logAPPL(3)

2. Edit **/usr/spool/log/formats/APPLmsg** file and replace the custom error definitions with place holders.

   In the example in Chapter 7, "Application Example" 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.
   ```

   Should be replaced with:

   ```
   APPL001  -- -- --- ({MNEMONIC}) %s
   APPL002  -- -- --- ({MNEMONIC}) %s
   APPL003  -- -- --- ({MNEMONIC}) %s
   ```

3. Rebuild the errors file by entering the following:

   cd /usr/spool/log/formats
   make -f formats.mk install
   reinitLog

   This removes your custom errors from the log.

Upgrade Considerations

6

What's in This Chapter

NOTE:
This chapter assumes a knowledge of the application development environment, the C language, and requires file editing, possible application source modification, and use of the provided conversion tools. Do *NOT* attempt upgrades unless you feel you meet these requirements.

This chapter details the following upgrade considerations:

- Upgrading message handling in data interface processes (DIPs) that interfaced with the Error Tracker (ET) mechanism to the current logging environment
- Upgrading transaction assembler (TAS) script and DIPs. Both require recompilation for Intuity CONVERSANT Voice Information System (VIS) Version 5.0.

NOTE:
If your application does not interface with ET (that is, if you have a data interface process [DIP] and it does not contain any **et_send** calls), or if you have no DIPs or tas scripts, the procedures need not be performed.

These procedures also assume that you have followed the documented procedure for writing a DIP process, that is, you have placed your message number defines in a header file named **appl_et.h**, etc. If you have expanded upon or deviated from the recommended procedures, you will have to understand shell programming and perform additional steps to complete the upgrade procedure.

Upgrading Message Handling in DIPs

The following procedures detail how to upgrade the message handling of your native script or DIP to work with the Version 5.0 environment. These procedures are meant to be performed sequentially.

Saving Explain Text

The following procedure must be performed *prior* to upgrading a custom application if custom explain messages have been created for the application:

1. Before the upgrade, enter the following command:

   > appl.explain
   while read errNum ; do explain $errNum >>appl.explain 2>&1 ;
   done
   {type each error number defined in "appl_et.h", that has
   explain text associated with it, 1 per line, end with <CTRL>D}

   This produces the file **appl.explain** that has the form:

   The message for error code 101 is:
   A serious system error has occurred.
   Reboot the system.
   The message for error code 102 is:
   A fairly serious system error has occurred.
   If it happens again, reboot the system.
   The message for error code 103 is:
   This error is not serious at all. However, if it
   happens three times a week for more 10 consecutive weeks,
   reboot the system.

2. Save the **appl.explain** file that contains the explain messages on external media (for example, on a floppy disk). Also, save **appl_et.h** (or whatever header files contain the programming defines) for the application error numbers defined by your application.

Restoring Explain Text

1. After the current release has been installed, restore the **appl.explain** and **appl_et.h** files.
2. Create the following shell script:

   ```
   cat >/vs/upgrade/fmtExp <<!
   vName=\‘basename \$3\‘ ; echo ’<< ’\$vName’ >>’ ; echo ; cat \$3

   chmod +x /vs/upgrade/fmtExp
   ```

3. Enter the following command:

   ```
   /vs/upgrade/upgExp
   ```

   You are prompted for the name of the file with your explanations in it and the name of the header file with the defines. If your names match the default file names (that is, **appl.explain** and **appl_et.h**), press ENTER. Otherwise, type the pathname for the requested file.

   If all of your error number defines are less than or equal to 14 characters in length, **upgExp** will automatically convert each explanation into the form appropriate for Version 5.0. A session might look like:

   ```
   /vs/upgrade/upgExp

   Name of the file in which the output from the pre-3.1 explain
   for the application DIP error numbers was saved: [appl.explain]
   Name of the header file defining the error numbers: [appl_et.h]
   Converting 109  APPL001 CORRUPT_DATA
   Converting 110  APPL002 DATA_TOO_OLD
   Converting 111  APPL003 NO_SERVER
   ```

   If any of your define names are 15 or more characters in length, upgExp will suggest a *short* name, truncating after the 14th character. If you are satisfied with this short name, press ENTER. This *short* name is the internal file name in which the explanation will be stored. The full *long* mnemonic is still an acceptable way to reference the explanation.

   If an explanation is found in your explanation text file, **appl.explain**, but is not found in your header file, **appl_et.h**, **upgExp** will ask you for the mnemonic name associated with the error number.

   You may rerun the **upgExp** command on the same set of explanations if you decided that you would like to change the actual explanation texts. First edit the explanation text file, **appl.explain**, changing *only* the text of the explanation itself (the lines of text between those beginning with “The message for error code...”). Once you have the explanations as you want them to appear, type **upgExp** again. The second and subsequent times, you will be asked if you want to overwrite each explanation. Press ENTER.

   You have completed this procedure.

Upgrading an Application's Message Handling

When upgrading an application, two types of conversion processes are provided:

- A full conversion

  Of the two conversion processes, a full conversion is the recommended conversion. This process is the most efficient in terms of execution speed and provides the most features to application developers. It also produces logged messages which match the new logging message formats. Also, the user interface will provide the means to adjust message priorities, destinations, and thresholds (see Chapter 3, "Configuration Management," of *Intuity CONVERSANT VIS V5.0 Operations,* 585-310-550).

  This method removes all **et_send** calls from C-source files of the application being converted. Consequently, when the application is executed, any log messages generated will be sent directly to the new logger process (logdaemon).

- A transparent conversion

  Use this type of conversion only when applications must work in pre-3.1 release environments as well as the current release. Support for this transparent conversion will be discontinued in future releases.

  This method allows applications whose C-source files contain **et_send** calls to continue to work and to log error messages without any changes.

Full Conversion

To complete a full conversion in an upgrade process, follow the steps below.

1. Copy to tape or diskette all of the application source code, headers, and **/gendb/data/errors** file which were saved before the older VIS version was removed.

   (See the tables in Chapter 3, "Manual Software Upgrades," of *Intuity CONVERSANT VIS V5.0 Upgrade,* 585-310-152, which contain steps to insure this.)

2. After upgrading the machine to the current release, restore the application source code, headers, and **/gendb/data/errors** file to the machine.

3. Perform the following steps to create the file **/usr/spool/log/head/logAPPL.h**. This file should *not* exist prior to the upgrade procedure.

   ```
   cd /usr/spool/log/head
   if diff ORGlogAPPL.h logAPPL.h
   then rm logAPPL.h
   else echo You will need to manually combine o.logAPPL.h after upgrade.
   rm -f o.logAPPL.h
   mv logAPPL.h o.logAPPL.h
   fi
   ```

4. Remove all the comment lines up to the first "#include" at the top of file **/gendb/data/errors**.

5. If your error numbers are described in **/gendb/data/errors** file as they should be and if the name of your header file defining your error numbers is **/att/msgipc/etmsgs/appl_et.h**, enter the following command:

   /vs/upgrade/cvtAPPL

   This shell script creates two files, **/usr/spool/log/formats/APPLmsg** and **/usr/spool/log/head/logAPPL.h**. The former file contains the formats of each error number message. The latter file contains the mnemonic defines for each error number to be used by your application code. If you used a header by some other name, you must either temporarily rename it **appl_et.h** so that you can use the cvtAPPL script, or you can manually execute the commands contained in **/vs/upgrade/cvtAPPL** with appropriate modifications to perform the necessary upgrade procedure.

6. Enter **cd /usr/spool/log/formats**

7. Enter **make -f formats.mk install**

8. Verify that:

   - **APPLmsg** exists and contains the message formats for your applications
   - The file **../head/logAPPL.h** exists and contains the define symbols identifying your logging messages
   - **cmpLogFmt**, **textLogFmt.Mne**, **textLogFmt.NoM** exist in . and ..
   - **textLogFmt** exists in **..** and is linked to either **textLogFmt.NoM** or **textLogFmt.Mne**
   - **systemLog.h** exists in both **.** and **../head**

9. Enter **msgadm** as described in the *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230, or use the message administration screen to reestablish the priority of each APPL message and specify the destination as described in Chapter 3, “Configuration Management,” of *Intuity CONVERSANT VIS V5.0 Operations,* 585-310-550.

NOTE:
This step is optional if full pathnames were used in the previous step.

10. Enter **cd *{src-directory}*** where *{src-directory}* is the directory in which your application code is located.

11. Create the file **/tmp/cvt.lst** files, containing a list of all compilable source code, ***.c**, ***.h**, and ***.pc** files, for any applications programs you want to recompile for the current environment.

    Each file name should appear on a line by itself. Pathnames should either be full pathnames (starting with '/') or relative to the directory in which you plan to run the upgCode program.

    NOTE:
    If you have not used **appl_et.h** as the name of your header file defining your error message numbers, you will need to amend the rules used to upgrade header files before proceeding. Your header file should have been named something like xxx_et.h.

12. Having already performed the header file upgrade in Step 3, you should now have a new header file of the form, **log*XXX*.h**.

    To amend the rules:

    a. Edit the file **/vs/upgrade/incChgs**.

    b. Add a line of the following form to the end of the list of rules:

       xxx_et.hh'+.3i'logXXX.h

       where "*xxx*" and "*XXX*" are the specific file name identifiers you used in your application.

13. Enter: **/vs/upgrade/upgCode**

14. Examine all files for which warning messages were created during the conversion process. The list of potential problems is in **/tmp/cvt.lst**.

    Problems fall into three major classes:

    a. Use of #define symbols or structure definitions from the header files **et_send.h** or **et_.h**

       For example, references to struct et_msg, ET_MSG, and ERRORS_ATT will generate warnings. The complete list of items that will produce warnings is found in **/vs/upgrade/warningPats**. Manually change the code so that it no longer uses these symbols or structure elements.

b. Calls to **et_send** which are expressed as a macro

For example:

```
#define ETSEND(msgID,arg,str)
et_send(chan,msgID,arg,0,0,0,str)

ETSEND(VMD_MMLUPD,5,"message text")
```

The conversion program cannot deal with such cases. To allow the conversion program to operate on the previous example, manually change it to the following form:

```
et_send(chan,VMD_MMLUPD,5,0,0,0,"message text")
```

c. Calls to **et_send** which use a variable for the message ID

For example:

```
void func(id,arg0,arg1,str)
int id,arg0,arg1 ;
char *str ;
{
et_send(-1,id,arg0,arg1,0,0,str) ;
}
```

These are the most complicated to convert. Read the manual page for the **logMsg** function in Appendix B, "Voice System C-Library Functions"

The **logMsg** function replaces the **et_send** function. Where **et_send** had an invariant set of arguments in a fixed order, **logMsg** has a variable list of arguments whose order and type depend on the message being logged.

To convert such references, you will have to understand the true intent of the underlying code. In the example above, the best solution would be to just replace the calls to func() with direct calls to **logMsg** since this function does not do anything except log a message.

In more complicated cases, the solution will depend on the format of the messages that must be logged. Make changes to the code to resolve the problems if necessary.

15. Add a call of the form:

logInit("{*program-name*}")

to each executable.

{program-name} is the name you want associated with messages logged by your application (for example, DBDIP). This name should match the bulletin board (BB) name provided to the **startup** or **VSstartup** routine.

You should normally place this call in the main() routine. This call must appear prior to any attempt to log information via the **logMsg**, **vlogMsg,** or **logSysError** functions.

16. Add the compiler option:

    -I /usr/spool/log/head

    so that headers files can be referenced from the **/usr/spool/log/head** directory.

    You may remove the compiler options

    -I /att/msgipc/etmsgs

    if you so desire, since header files from the old system will no longer be used after conversion.

17. Add the following libraries to the make rules file of the application to include them in the load option:

    /vs/lib/liblog.a
    /vs/lib/libprism.a

18. Recompile all modified applications.

    You have completed this procedure.

Transparent Conversion

1. Restore **/gendb/data/errors** file which was saved earlier. Install this file in the same directory on Version 5.0.
2. Enter **/vs/bin/mkerr var**
3. Verify the file **/vs/data/etStub.rules** was created.
4. Recompile and install the application executables in the VIS V5.0 machine.

Backward Compatibility

The command **into_et** was used prior to the Version 3.1 release to log messages from a DIP or application. **into_et** still exists in the system but it only works on those messages that are *not* using the current logging environment. That is, you can only use **into_et** with messages in the pre-Version 3.1 format. The commands **logMsg** and **logEvent** now are available to provide this functionality

Upgrading an Application

The following procedures detail upgrading your TAS application script and DIP for use in the Intuity CONVERSANT VIS V5.0 environment.

Compiling the TAS Script

In Intuity CONVERSANT VIS V5.0, the TAS compiler now allows the TAS language to be enhanced more easily and to provide better error checking and reporting. If you have upgraded your system to V5.0 from a previous release, TAS V5.0 may have been installed with "4.0 Compatibility Mode" set. A message to that effect comes out each time the **tas** command is run if this is true. Compatibility Mode allows older scripts that may not meet the TAS language definition to compile with a minimum of changes.

Script Changes for Compatibility Mode

The following script source code changes may be required before TAS 5.0 will successfully compile a script in Compatibility Mode. These are known incompatibilities that should occur only rarely. These incompatibilities produce error messages that prevent successful compilation of the script.

NOTE:
Warning messages may also be produced by TAS in Compatibility Mode. These messages do not prevent successful compilation, but indicate changes that should be made in the script source code to insure compatibility with future releases of Intuity CONVERSANT.

- Invalid register number errors

  Script instruction arguments that require register numbers now have range checking done on them by TAS (for example, arguments of the form r.*X*, *ch.*X*, *int.*X*, *sh.*X*, *ev.*X* -- where *X* must be a script register number). In releases previous to V5.0, only 4 registers were available, so *X* should be in the range 0–3. In Intuity CONVERSANT VIS V5.0 there are 16 registers available, so the valid range for *X* is 0–15. Invalid register numbers are a script bug and should be corrected.

  The most common mistake with the argument types that begin with '*' is not that the register number is wrong, but that the '*' was put there by mistake. For example, the argument "int.254" refers to the integer value stored at script address 254. The argument "*int.254" refers to the integer value stored at the script address contained in register 254. In the latter example, 254 is not a valid register number. Prior to 5.0, TAS would not catch this error and TSM would substitute a null value for the argument when executing the script. Therefore, in cases of arguments like "*int.254", it is likely that the '*' was put there by mistake (especially if the number is not even close to the valid range for register numbers).

For errors involving an argument of the form **r.*X*** (where *X* is negative or greater than 15), the error is either in the use of a register value where some other data type was intended or in the use of an invalid register number. Examine the context of the instruction to determine the programmer's intent and make the correction accordingly.

- An instruction argument with a missing '.' character between immediate data type keyword ("im", "imm", "immed", etc.) and a quoted string (for example, "im"xyz"")

  Place a dot between the data type and the quoted string or delete the data type keyword altogether since it is optional in this case (that is, im."xyz" and "xyz" are equivalent). Prior to Intuity CONVERSANT VIS V5.0, TAS would accept this syntax (leaving out the dot in any other argument type was not acceptable). The error messages produced by this syntax in Version 5.0 indicate that the instruction as the wrong number of arguments as it is interpreted by the TAS compiler as two separate arguments (for example, a label argument and a literal string argument). This may also cause an "undefined label 'im'" message to be produced.

- Numeric labels

  Prior to Version 5.0, it was possible to define an unused label name with a numeric string rather than alphanumeric and TAS would not complain. Valid labels must always begin with an alphabetic character to distinguish them from integer constants. This usually happens in a script where a #define symbol that has an integer value is used. If a label happens to have the same name as the #define symbol, the integer value of the #define symbol will be substituted for the label name by the preprocessor before being run through TAS. The following script fragment illustrates this:

  ```
  #define VALUE 128
  ...
  load(r.0, int.VALUE)
  ...
  VALUE:
  ...
  ```

  When this script is run through the preprocessor by TAS, the value 128 is substituted for every occurrence of VALUE. This is acceptable for the load instruction (which is loading an integer at script address 128 into register 0), but the label is changed to "128:"; which is an invalid label definition.

  If a script like this compiled with the old TAS, then it is certain that the label was not being used anywhere in the script (for example, by any **goto()** or **jmp()** instructions) since instructions do not allow integer constants as labels. Therefore, the offending label should be deleted to fix the problem.

- Deleted instructions

  The following instructions have been obsolete since CONVERSANT 2.1 or before and are no longer accepted by TAS in Version 5.0:

  — The **lic** instruction was used perform a function similar to the **tic** instruction. If your scripts contain any **lic** instructions, change them to **tic** (the arguments are the same; see Chapter 3, "Script Instructions" or Appendix A, "Summary of Script Instructions".

  — The **phcreate** instruction was replaced by the **phreserve** instruction in CONVERSANT 2.1. If you have any of these instructions in your script, change their names to **phreserve**, convert the third argument from kilobytes (Kb) to seconds (if necessary), and add a 4th argument ADPCM32 to the instruction. For example, change

  ```
  phcreate(im.-1, int.TALKID, im.-1)
  ```

  to:

  ```
  phreserve(im.-1, int.TALKID, im.-1, im.ADPCM32)
  ```

  If the 3rd argument to **phcreate** was not 0 or -1 (as above), then that argument value convert from Kbs to seconds by dividing the old value by 4. For example, change

  ```
  phcreate(im.-1, int.TALKID, im.40)
  ```

  to:

  ```
  phreserve(im.-1, int.TALKID, im.10, im.ADPCM32)
  ```

  The 4th argument specifies the coding type and rate. ADPCM32 is an integer define symbol in the codestyle.h header file so the line

  ```
  #include “codestyle.h”
  ```

  should be added at the beginning of the script file containing the **phreserve** instruction if it is not already there. ADPCM32 was the assumed coding type for the old **phcreate** instruction. If you like you may use another supported type from the list in the **/att/include/codestyle.h** file. See the **phreserve** instruction in Chapter 3, "Script Instructions" and Appendix A, "Summary of Script Instructions".

Turning Off Compatibility Mode

If TAS was installed with Compatibility Mode set, you can unset the mode by renaming the file **/vs/data/tas.debug**. For example, use the UNIX command **mv /vs/data/tas.debug /vs/data/tas.debug.save**. To set the mode back, move the saved file back to its original name.

NOTE:
If the file gets deleted, simply create the **/vs/data/tas.debug** file with the contents "0x2000" to set Compatibility Mode.

Running TAS Without Setting Compatibility Mode

Running TAS without Compatibility Mode set may produce more warning and error messages for a particular script. These are problems that should be corrected in the script source code to ensure compatibility with future releases of Intuity CONVERSANT. All new scripts you develop should be compiled without Compatibility Mode. Existing scripts that have been compiled in Compatibility Mode should be modified at some point (if necessary) to compile without Compatibility Mode.

Error and warning messages are meant to be self-explanatory. They point the user to the file, line number, instruction, and argument where the error or warning is caused and give a brief explanation of the problem. Error messages prevent successful compilation of the script by indicating improper TAS language syntax.

Warning messages do not prevent successful compilation but indicate that TAS is accepting an outdated form of an argument or instruction which may not be acceptable in the future. These messages indicate what the acceptable form should be so that the source code can be changed accordingly to remove the warnings. The **-w** command line option may be used to suppress all warning messages. This may be useful to better see the error messages if they are being overwhelmed in the output by numerous warning messages.

Compiling the DIP

When upgrading to Intuity CONVERSANT VIS V5.0, all DIPs must be recompiled. Be aware that the V5.0 compiler is likely to report errors in your DIP (for example, due to lack of function prototyping in your existing DIP). Refer to Chapter 4, "Data Interface Process" for the procedure to compile your DIP. If additional information is required, consult any standard ANSI-C reference document.

Application Example

7

What's in This Chapter

This chapter presents an example of an application. It includes:

- Two versions of the script

  The first version shows the Script Builder action steps and the second shows the script instructions generated by the Script Builder software.

- 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 example 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 Intuity CONVERSANT Voice Information System (VIS).

Sample Script — Script Builder Action Steps

The following example shows the action steps defined in Script Builder.

```
 start:
1. Answer Phone
2. Announce
     Speak With Interrupt
     Phrase: "Hello, this is the DIP test script"
   entry_loop:
3. Set Field Value
     Field: acct_balance = 0
4. Prompt & Collect
     Prompt
       Speak With Interrupt
         Phrase: "Please enter your SSN"
       Input
         Caller Input Field: ssn
         Min Number Of Digits: 09
         Max Number Of Digits: 09
     Checklist
       Case: "Input Ok"
         Continue
       Case: "Initial Timeout"
         Reprompt
       Case: "Too Few Digits"
         Reprompt
       Case: "No More Tries"
         Quit
     End Prompt & Collect
5. Prompt & Collect
     Prompt
       Speak With Interrupt
         Phrase: "Please enter your account number"
     Input
       Caller Input Field: acct_num
       Min Number Of Digits: 06
       Max Number Of Digits: 06
     Checklist
       Case: "Input Ok"
         Continue
       Case: "Initial Timeout"
         Reprompt
       Case: "Too Few Digits"
         Reprompt
       Case: "No More Tries"
         Quit
   End Prompt & Collect
6. External Function
       Function Name: diptest
       Use Arguments: ssn acct_num
       Return Field: acct_balance
```

```
7. Evaluate
   If acct_balance < 0
8.  Announce
    Speak With Interrupt
      Phrase: "This is an error situation, the return value
      is"
      Field: acct_balance As Nrmf
9.  Quit
    End Evaluate
10. Set Field Value
    Field: acct_balance = acct_balance
11. Announce
      Speak With Interrupt
        Phrase: "Your account balance is"
        Field: acct_balance As N$
12. Prompt & Collect
    Prompt
      Speak With Interrupt
        Phrase: "Enter 1 to enter ssn again, 2 to quit"
    Input
      Caller Input Field: prompt
      Max Number Of Digits: 01
    Checklist
      Case: "1"
        Goto entry_loop
      Case: "2"
        Continue
      Case: "Not On List"
        Continue
      Case: "Initial Timeout"
        Reprompt
      Case: "Too Few Digits"
        Reprompt
      Case: "No More Tries"
        Quit
   End Prompt & Collect
13.  Announce
      Speak With Interrupt
        Phrase: "Thank you for calling the Dip Test Script"
14.  Disconnect
15.  Quit
```

Sample Script — Script Language

The following example shows the same script in the script language described in Chapter 3, "Script Instructions". This script was generated by Script Builder. The script labels used by Script Builder are not as descriptive as labels that a programmer would use.

```
/* 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 SZUV          17      /* 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 except that the DIP has been modified to manipulate the social security number as described in the introduction to this chapter.

```
#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
```

Summary of Script Instructions

A

What's in This Appendix

This appendix contains information about the script instructions discussed in Chapter 3, "Script Instructions" and Chapter 4, "Data Interface Process" of this book. Typically, the information here is the same as in Chapters 3 and 4, but is presented in a different format.

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

- Instruction name and syntax
- Purpose of the instruction
- Effects of using the instruction
- Examples of the instruction

Script Instruction Syntax

In presenting a script instruction's syntax, 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 (a|b|c|d).

For more information about the conventions used in this book, refer to "Conventions Used in This Book" in the "About This Book" section.

and

Name

This script 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,im.FLAG)
```

atoi

Name

This script 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

This script instruction starts and/or listens to background audio on the specified channel.

Synopsis

background(*"phrase_name",type.src*)
background(*type.src,type.src*)

Description

NOTE:
A time division multiplexor (TDM) bus and signal processor (SP) circuit card must be installed in the system for the background instruction to function properly.

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 VIS, 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 along 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.

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:

- Attempt to add more than one background audio to a channel
- Channel reached the limit for listen time slots (maximum of 7 per channel)
- No SP circuit card is available
- All TDM slots are busy
- Reached system limit on number of backgrounds (MAXCHAN)
- System call failure

NOTE:
On a T/R channel that is not using the TDM bus to play speech (for example, the channel is set to “talk”, not “tdm”), the foreground speech interrupts background speech. If the TDM bus is used, background speech is heard continuously.

Example

```
#define ADD 1
#define DROP 0

tfile("/speech/talk/list.cabnt")
background("begin testing",imm.ADD)
background(imm.201,imm.DROP)
```

case

Name

This script 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 phone numbers is dialed.

```
case(int.FOLLOW_UP,im.F_ATD_A,CALL(im.APHONE),w4answ)
case(int.FOLLOW_UP,im.F_ATD_B,CALL(im.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 A-1. chantype Return Values

Value	Channel Type
IRD_TR	Tip/Ring
IRD_T1	T1 E&M protocol
IRD_PRI	IDSN PRI protocol
IRD_LST1_DEF	Line Side T1 for DEFINITY
IRD_LST1_GAL	Line Side T1 for Galaxy
IRD_LST1_ASAI	Line Side T1 for DEFINITY with ASAI
IRD_ASAI	Tip/Ring with ASAI
IRD_VIRT_CHAN	Virtual channel

A negative value is returned if an error occurs.

Example

For example:

```
#include "/att/include/irDefines.h"

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


/* channel type must be TR or 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)
```

dbase

Name

This script instruction sends a message to a data interface process (DIP).

Synopsis

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

Description

Dbase 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. Refer to Chapter 4, "Data Interface Process" 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(im."Bankdip",LOGON,ch.CUSTRECS,SZCUSTREC,ch.LOGONS,5)
```

See Also

getdig

decr

Name

This script 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,im.NSTKS)
```

dipname

Name

This script 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,imm.0) /* DIP number 0 */

dipname(ch.DIPNAME,int12) /* DIP number in integer location 12 */
```

See Also

dipnum

dipnum

Name

This script 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:

1. Is invalid
2. Does not have an associated message queue already created
3. 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, imm."Dip")
```

See Also

dipname

dipterm

Name

This script 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 A-1 and Figure A-2 show the fields of the message and their values as set by TSM.

```
/* message structure for dipterm message */
struct ms_univ {
     struct mbhdr  hd;
     long          arg[4];
};
```

Figure A-1. dipterm Synopsis

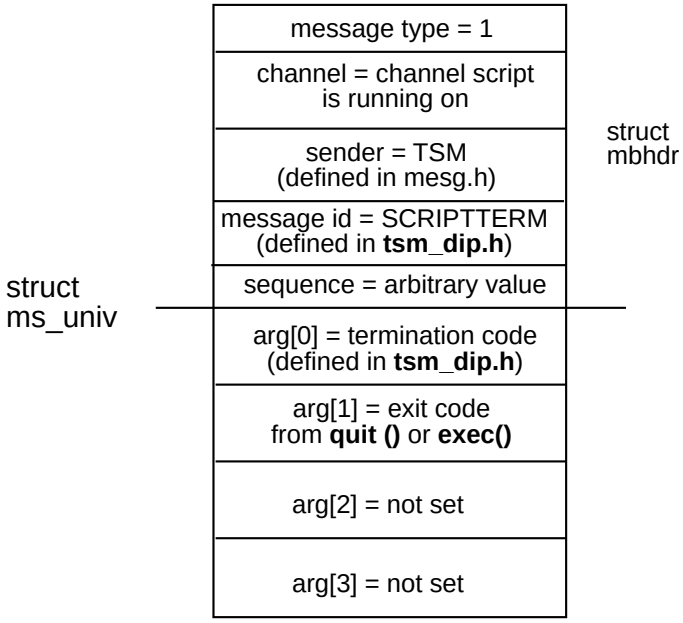


Figure A-2. dipterm Message Structure

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 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.

Example

The following example causes DIP0 to receive a termination message when the script terminates.

```
dipterm(im.0)
```

The following example causes the DIP called “bankdip” to receive a termination message when the script terminates.

```
dipterm(im."bankdip")
```

div

Name

This script 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,im.NSTKS)
```

dtitos

Name

This script 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 UNIX system representation to “tm” structure form. The *type.src* argument should contain a number representing the UNIX 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 9-integer structure defined in CTIME(3C) in the *UNIX System V 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_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 WKDAYPH      44
#define MONTHPH      48
```

```
tfile("stdspch.pl")
dtitos(time.0, ch.TM)
tic('a')
talk("date")
load(int.WKDAYPH, im."sunday?")
incr(int.WKDAYPH, int.TM_WDAY)
talk(int.WKDAYPH)
load(int.MONTHPH, im."january")
incr(int.MONTHPH, int.TM_MON)
talk(int.MONTHPH)
tnum(int.TM_MDAY)
tnum(imm.19)
tnum(int.TM_YEAR)
sleep(im.2)
talk("time")
tnum(int.TM_HOUR)
tnum(int.TM_MIN)
sleep(im.2)
talk("goodbye")
quit()
```

See Also

dtstoi

dtstoi

Name

This script instruction converts the date and time from the “tm” structure to a UNIX 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 UNIX 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, this script fragment gets the current system date and time, truncates the hour to midnight, and converts the result back to UNIX 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, imm.0)
dtstoi(ch.TM, int.MIDNGT)
```

See Also

dtitos

event

Name

This script 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 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 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 was 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 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 was used.

- EDIPINT — DIP interrupt event. This event may 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 is ignored and the script continues to run.

- ETTREC — 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 (“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 — Answer supervision event. This event is triggered when answer supervision is detected for a 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.

The DIP number stored in r3 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 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, im.-1)            /* reset event */

     ...

L_tkoff
     jmp(r.3 !=im.'t', L_notkoff)        /* not tflush */
     jmp(r.1 !=im.'#', L_notkoff)        /* not # digit */
     tstop()/* stop play */
     load(r.0, im.-5)   /* abort tflush() */
     rts()
L_notkoff:
     load(r.0, im.0)  /* continue instruction wait */
     rts()
```

exec

Name

This script 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 which 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 insure 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(im."test.script",1)
```

See Also

dipterm
execu
quit

execu

Name

This script 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, intact, the data space of the "parent" script. 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(im."test.script",1)
```

See Also

exec

getdig

Name

This script instruction receives touch-tone or spoken word data from a caller.

Synopsis

getdig(*type, ctype.dst, number,[type.mode])*

Description

The **getdig** instruction receives information entered by a calling party using touch tones or voice input. Type 0 specifies touch-tone input.

The number argument specifies the maximum number of touch-tone digits to be received. Received touch tones are stored as a null terminated character string in a buffer specified by the destination argument.

Getdig has a 10 second default initial digit wait time for touch-tone input. If the caller does not enter a touch tone within the allotted time period, getdig returns the number of digits received before the timeout occurred. Use the tttime command to specify desired wait times.

Getdig is a wait-causing instruction. Therefore, it automatically forces out any pending or unfinished announcements from this channel.

When this instruction terminates, a return code is placed in r.0. The following list shows the return values for touch tone input, where *N* represents the number of touch tones received:

N > 0	If the *number* argument is greater than 0 (fewer than the expected number of touch tones were received), an interdigit time out occurred.
N = 0	An initial timeout occurred.
N < 0	A system error occurred during the playing of a prompt or the getdig instruction itself.

Example

In the following example, the script waits for the caller to enter up to ten touch tones, then stores them in ch.ANS.

```
getdig(0, ch.ANS, 10)
```

Feature Related Changes

The following information applies to the getdig instruction when using it for WholeWord and FlexWord speech recognition. This information supplements that provided earlier about the **getdig** instruction.

Using getdig with WholeWord Speech Recognition

The first argument, *type*, specifies whether touch-tone or speech input is expected from the caller. Type 0 specifies 12-key telephone touch-tone input. A non-zero value for *type* specifies speech input. The **getdig** instruction requires the recognition type used for a particular grammar. The choices available for *type* in this instruction can be found in the file **/att/include/sr_grammar.h**. The grammars and their specifying values are listed in *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228.

NOTE:
For packages which support connected-digit recognition: For US English, you may use US_0_9o to recognize any variable-length string of 1-24 digits. If the string length is known in advance, however, superior recognition performance can be obtained by using one of the grammars with a fixed string length.

If the *type* argument is 0, the *number* argument specifies the maximum number of touch-tone digits to be received. The maximum value is 128. Received touch tones are stored as a null terminated character string in a buffer specified by the destination argument, *dst*.

If the type argument is other than 0, the number argument specifies the maximum string length for speech input. Received speech input is stored as a null terminated character string in a buffer specified by the destination argument. The characters are defined by the vocabulary provided. Possible characters are listed in *Intuity CONVERSANT VIS V5.0 Speech Development*, 585-310-228.

NOTE:
For packages which support connected-digit recognition: The maximum value of the string length for speech input is 24.

The fourth argument, *type.mode*, indicates to the script whether the response is touch-tone or voice. If the response is touch-tone, -1 is stored in *ctype.mode*. If the response is voice, then the number (non-negative) of the SP circuit card that recognizes the voice is stored in *ctype.mode* to be used later by a DIP.

When the **getdig** instruction terminates, a return code is placed in r.0. The return values for touch-tone input and speech input are listed in the following tables:

Table A-2. Return Values for Touch-Tone Input

N > 0	If the *number* argument is greater than 0 (fewer than the expected number of touch tones were received), an interdigit time-out has occurred.
N = 0	An initial time-out has occurred.
N < 0	A system error occurred during the playing of the prompt or the getdig instruction itself.

Table A-3. Return Values for Speech Input

N > 0	Speech was heard and recognized. In this case, *N* represents the length of the string containing the word recognized.
N = 0	No valid speech was heard by the system. (An initial time-out has occurred.)
N < 0	A system error occurred during the playing of the prompt or the getdig instruction itself.

getdig Example for WholeWord Speech Recognition

In the following example, the script accepts the spoken digits 1 through 3 or the word “no” in a string no longer than the length defined in im.01. It stores the received input in ch.F__CI_VALUE. It stores the indication of speech or touch-tone input in int.F__CI_MODE.

```
getdig (US_1_3n, ch.F__CI_VALUE, im.01, int.F__CI_MODE)
```

Using getdig with FlexWord Speech Recognition

The first argument, *type*, specifies whether touch-tone or speech input is expected from the caller. Type 0 specifies 12-key telephone touch-tone input. A non-zero value for *type* specifies speech input. The getdig instruction requires the recognition type used for a particular wordlist. The choices available for *type* in this instruction can be found in the **/vs/data/sr_file**. The sr_file contains a line for every wordlist installed on your system. If WholeWord is installed, this sr_file will also list all of the grammars on the system. Information in this file will map wordlist names into the symbols that you need to use with this instruction.

Therefore, for a specific wordlist, look to the fourth field and use this "defined symbol," which correlates with the wordlist on the same line.

If the *type* argument is 0, the *number* argument specifies the maximum number of touch-tone digits to be received. The maximum value is 128. Received touch tones are stored as a null terminated character string in a buffer specified by the destination argument, *dst*.

If the type argument is other than 0, the number argument specifies the maximum string length for speech input. Received speech input is stored as a null terminated character string in a buffer specified by the destination argument. The characters are defined by the words in the wordlist.

The fourth argument, *type.mode*, indicates to the script whether the response is touch-tone or voice. If the response is touch-tone, -1 is stored in *type.mode*. If the response is voice, then the number (non-negative) of the SP circuit card that recognizes the voice is stored in *type.mode* to be used later by a DIP.

When the **getdig** instruction terminates, a return code is placed in r.0. The return values for touch-tone input and speech input are shown in Table A-2 and Table A-3.

getdig Example for FlexWord Speech Recognition

In the following example, the script words from, for example, the SAVINGS wordlist in a string no longer than the length defined in im.64. It stores the received input in ch.F__CI_VALUE. It stores the indication of speech or touch-tone input in int.F__CI_MODE.

```
getdig (WL_11, ch.F__CI_VALUE, im.64, int.F__CI_MODE)
```

See Also

ttclear
ttdelim
tflush
tttime

goto

Name

This script 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 would jump 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 <= im.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

This script instruction directs the current channel to bridge partially to another channel.

Synopsis

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

Description

The **hbridge** script 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 OTHCHAN   17

hbridge (imm.OTHCHAN,imm.ADD)
hbridge(imm.OTHCHAN,imm.DROP)
```

hundsec

Name

This script 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

This script 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*, then 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

This script 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,im.1)
```

itoa

Name

This script 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

This script 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==imm.'*',attendant)
jmp(*ch.0==imm.'#',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(in.FIRST,im.1)
load(in.SECOND,im.2)
ADDEM(in.FIRST,im.SECOND)
tnum(in.SUM)

ADDEM( )
   load(in.SUM,imm.0)
   incr(in.SUM,r.3)
   incr(in.SUM,r.2)
   rts ( )
```

See Also

case

listenall

Name

This script 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 (imm.45, imm.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, "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 DROP      0
#define OTHCHAN     17

listenall(imm.OTHCHAN,imm.ADD)
listenall(imm.OTHCHAN,imm.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,im.NSTKS)
load(r.2,im.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,im.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(im.15)
```

See Also

event
sleep

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,im.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(imm.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,im.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,im.19)
phremove(im.205,int.TALKID)
```

In the following example, phrase 117 is removed from talkfile 10.

```
load(ch.TALKF,im.10)
load(int.PHR,im.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,im.15)
phreserve(im.-1,int.TALKID,im.-1,im.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,im.8)
phreserve(ch.20,im.23,im.10,im.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(im.0)
quit( )
```

See Also

dipterm
exec

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 VIS 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 A-4. 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 SP 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(im.”For everything there is a season, “)
say(im.”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()
```

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(im."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. This instruction can be used without first using the **tfile** instruction. 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 previous talkfile id is stored at location int.OLDTALK. The phrase number 210 spoken by the **talk** instruction refers to the speech phrase encoded in talkfile 25 and not to the speech phrase listed in list.cabnt.

```
tfile("/speech/talk/list.cabnt")
setalk(imm.25)
load(int.OLDTALK,r.0)
talk(imm.210)
```

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

Example

For example, to set an attribute that requests an SID for the Calling Party Number (CPN), use:

```
setattr (ATTR_SID_O)
```

setcca

Name

This script instruction sets the type of 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 for 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, Intelligent CCA is used. If mode is 1, Full CCA is used.
- *type.nrings* — Number of rings to wait for answer. This parameter can be between 1–10 rings.
- *type.ansdet* — Answer detection. 0 = no, 1 = yes. The default is -1 (yes for T/R and LST1 lines, no for T1 and PRI). By default, answer detection is turned on for T/R and LST1 lines and off for T1 and PRI lines because T/R and LST1 lines do not have answer supervision while T1 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'** or **tic'w'** instruction.

Example

In the following example, the call classification parameters are set to Full CCA, ten rings, and answer detection is enabled for T/R and LST1 lines and disabled for T1 and PRI lines.

```
setcca(im.1,im.10,im.-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(im.1,im.10,im.-1)
nextcall:
dbase( .... )     /* get number to dial from DIP */

tic('O', r.3)    /* call number in register 3 */

jmp(r.0 == im.'N', noAns)     /* no answer after 10 rings */
jmp(r.0 == im.'B', busy)
jmp(r.0 == im.'F', retry)
jmp(r.0 == im.'A', answer)
jmp(r.0 == im.'s', SIT)
jmp(r.0 == im.-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 == im.'R', retry)
jmp(r.1 == im.'r', retry)
jmp(r.1 == im.'K', retry)
jmp(r.1 == im.'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)
```

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 A-5. Valid Service Types

Service	Service Type
SDN (including GSDN)	SVC_SDN
MEGACOM 800	SVC_MEGACOM800
MEGACOM	SVC_MEGACOM
INWATS	SVC_INWATS
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

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(imm.SERVICE_TYPE, imm.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 types are:

Table A-6. Valid Bearer Types

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 SP 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(imm.SERVICE_TYPE, imm.SVC_NODAL_LDS)

     /* initialize character string to be dialed */
     strcpy(ch.DIALED_NUMBER, imm.”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,imm."6145551212")
```

Example 2:

```
load (int.STRINGTYPE imm.OUTBOUND_ANI)
strcpy (ch.CPN imm."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(im.0)` — Turn off touch-tone flushing
`setttfl(im.1)` — Turn on touch-tone flushing (default)

See Also

getdig
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(im.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

The **sp_alloc** instruction is explicitly used to allocate and deallocate speech recognition on the SP circuit card.

The **sp_alloc** instruction may be used by a script to allocate the speech recognition resource on the SP 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 SP 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 SP resource. If the resource is available, it is allocated to the script until the script terminates or until the script deallocates the SP resource using **sp_alloc**.

sp_alloc may be used to allocate an SP resource for a period longer than the script is actually recognizing speech. Care should be taken to avoid overloading the systems SP 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 SP resource

0 Deallocate the SP resource

The *type.resource* argument is used to tell the sp_alloc which SP resource or combination of SP resources to allocate or deallocate. Each SP 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 A-7. Resource Values:

1	Voice coding or playing
2	PRI function
4	WholeWord Speech Recognition
8	Call Classification
16	Text-to-Speech
64	Echo cancelling
256	FlexWord Speech Recognition

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 and FlexWord SP resources can be added together to allocate or deallocate more than one SP resource by using one **sp_alloc** instruction. See Table A-8 for examples.

Table A-8. sp_alloc Examples

Action	sp_alloc Script Instruction
Allocate WholeWord recognition resource for the script	`sp_alloc(im1,im.4)`
Deallocate FlexWord recognition resource for the script	`sp_alloc(im.0,im.256)`
Allocate both WholeWord and FlexWord recognition resource for the script	`sp_alloc(im.1,im.260)`

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. When speech recognition during prompt is enabled using **sr_talkoff(im.1)**, the **getdig** instruction starts playing any phrases in its queue and simultaneously turns on the recognizer. When speech recognition during prompt is disabled with **sr_talkoff(im.0)**, getdig 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 getdig. If a **tflush** instruction is used to initiate phrases immediately before the getdig, 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(im.1),	**getdig** plays phrases and *simultaneously* turns on the recognizer.
Speech recognition is disabled with sr_talkoff(im.0),	**getdig** plays phrases *then* turns on the recognizer.
tflush starts playing phrases before getdig,	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 card (T1, IVP4, or IVP6) that is connected to the TDM bus with the ‘‘tdm’’ option set. Enabling **sr_talkoff** requires that the SP circuit card play the prompts. The T1 and VRS cards are already set to ‘‘tdm.’’ However, for IVP4/6 circuit cards set to ‘‘talk,’’ the system detects a ‘‘recognition during prompt’’ request in the script and automatically uses an SP circuit card to play the prompts. If an IVP4/6 circuit card has the ‘‘talk’’ option set and **sr_talkoff** is off, the system plays all prompts with the IVP4/6 card. If an IVP4/6 circuit card has the ‘‘talk’’ option set and **sr_talkoff** is on, the system plays all prompts with the SP circuit card.

Because playing prompts uses resources on the SP circuit card, application designers who find that the SP resources are strained may want to consider configuring their IVP4/6 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(im.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 (imm."XYZ?:136",char.20)
jmp(r.0 ==imm.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 >= imm.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,imm."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 < imm.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(im."AB123,:=+")
jmp(r.0 !=im.9,Lab2)
```

talk

Name

This script instruction speaks one or more phrases.

Synopsis

talk("*phrase_name*"[,"*phrase_name*"]...)
talk(*type.src*)

Description

The first format for the **talk** instruction uses one or more arguments, each being a phrase in double quotes. Each phrase must match a phrase listed in the *application_name* file specified by the **tfile** instruction. It must also match according to the rules explained in Chapter 3, "Script Instructions". The second format uses the assigned phrase number which appears in the ***application-name*.pl** file or listfile.

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 either one of the following two conditions:

1. The script executes a speech-flushing instruction
2. The script executes a **say** instruction (Text-to-Speech)

Example

In the following example, the GREET subroutine is called on to say two introductory phrases from the talkfile specified in the **tfile** instruction. It then calls on a subroutine to speak the final phrase, “thank you.”

```
MAIN:
  tfile("/speech/talk/application_name.pl")
  GREET( )
  .......... ............... ............
  BYE( )

GREET:
  talk("hello," "please enter your id")
  indirect_talk(im."thank you.")
 ............ ............... ........
  rts( )
indirect_talk:
  talk(r.3)
  rts( )
```

See Also

tchars
tnum

talkresume

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(im.4)
```

tchars

Name

This script instruction speaks phrases from a variable name.

Synopsis

tchars(*ctype.src[,'inflection']*)

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 **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")
getdig(0,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 data base to use for the script. The first listfile name, called *application_name*.pl (see Chapter 2, "Development Guidelines"), 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**, **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**, **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 AT&T stock for the
price of")
tnum(int.SELL_PRICE)
talk("Enter your next instruction")
getdig(0,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',im.10,im.ADPCM32)
```

See Also

talk
tstop

tic

Name

This script instruction controls an attendant line.

Synopsis

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.

Tic 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, or PRI channel. Where such variations exist, they are noted below.

- *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 any call progress tone, then resume the script.
- *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 A-9 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: T/R, T1, PRI, or LST1.

Table A-9. Return Code Results for the tic Instruction

Meaning	Return Values		For tic	Available On			
	r.0	r.1		TR	T1	PRI	LST1
Instruction successfully completed	0	•	’a’,’d’,’f’*,’h’,’o’	•	•	•	•
Answer detected (e.g., voice energy detected or ringing stopped)	’A’	•	’W’,’w’†	•			
Answer supervision from switch	’P’	•	’D’,’O’,’W’,’w’		•	•	
Busy‡	’B’	•	’F’ud,’D’,’O’,’W’,’w’	•		•	
Fast busy (reorder tone)*	’F’	•	’F’ud,’D’,’O’,’W’,’w’	•		•	
Ring no answer	’N’	•	’W’,’w’	•			
Audible ringing	’R’	•	’F’,’D’,’O’	•			
Dialtone detected*	’D’	**	’F’,’D’,’O’,’W’,’w’	•		•**	
Stutter dialtone detected	’S’	•	’F’,’D’,’O’,’W’,’w’				
ISDN vacant code	’v’	**	(any)			•**	
Provisioning or protocol error	’p’	**	(any)			•**	
Internal hardware or software error or dialing error	-1	•	(all)	•	•	•	•

Continued on next page

Table A-9. Return Code Results for the tic Instruction — *Continued*

Timeout (no call progress tones detected within the timeout period)	-2	•	(all except 'h')	•	•	•	•
Illegal dial string passed††	-3	**	'D','O','d','o'	•	•	•**	•
Touch tone entry detected	't'	'O'	'O'	•	•	•	
Intercept tone heard representing an invalid extension (on DEFINITY or other AT&T PBX)	's'	'I'	'O', 'D'	•			

*A tic('F') or tic('f') instruction on a PRI channel will always fail (r.0 = -1).

†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') is done.

‡For PRI channels, the voice system converts certain information provided by the switch into busy, fast busy, or dialtone call dispositions. It does not necessarily mean that an audible tone is actually present.

**The disposition of calls on PRI channels is based solely on information provided by the switch. This particular disposition is not provided by the switch on every call. However when it is provided more specific info (the ISDN cause value) is available in register 1 (r.1). See Table A-11 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 A-10 for a list of r.1 return values.

††On a 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 channesl 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 T/R channel, 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 T1 and PRI, the maximum dial string is 15 digits. For T/R 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.

If your script will encounter a secondary dial tone, you may want to 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 VIS 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 VIS 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 VIS 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, imm."9999")
tic('O', ch.NUMBER)
jmp(r.0 == imm.-1, end)    /* hardware failure */
jmp(r.0 == imm.-2, end)    /* timeout, no response */
jmp(r.0 == imm.'B', end)   /* busy */
jmp(r.0 == imm.'R', ok)    /* ring */
end:
  quit()
ok:
  tic('h')
  rts()
```

Feature Related Changes

The following information applies to the tic instruction when using it in a Full CCA and a PRI application. This information supplements that provided earlier about the **tic** instruction.

Using tic with Full CCA

When an outbound application uses Full CCA, be aware that if Full CCA determines that the 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.

Full CCA Call Dispositions

Table A-10 lists the possible return values for the 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: T1, PRI, TR, or LST1.

Table A-10. Call Dispositions for tic('D') and tic('O')

Meaning	TSM Level		Available On:			
	r.0	r.1	TR	T1	PRI	LST1
Answer detected (for example, voice energy detected)	'A'	0	•	*	*	•
Answer supervision from switch	'P'	0		•	•	
Busy	'B'	0	•	•	•	•
Fast busy	'F'	0	•	•	•	•
Ring no answer	'N'	0	•	•	•	•
High and dry	'H'	0	•	•	•	•
Modem tone	'T'	0	•	•	•	•
Audible ringing	'R'	0	•	•	•	•
Dialtone detected†	'D'	0†	•	•	•	
Stutter dialtone detected	'S'	0	•	•	•	
ISDN vacant code	'v'	†			•	
Provisioning or protocol error	'p'	†			•	
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	-3	0†	•	•	•	•
CCA resource used up	-4	0	•	•	•	•
Reorder, intraLATA SIT	's'	'R'	•	•	•	•
Reorder, intraLATA SIT	's'	'r'	•	•	•	•
No circuit, intraLATA SIT	's'	'K'	•	•	•	•
No circuit, intraLATA SIT	's'	'k'	•	•	•	•
Vacant code SIT	's'	'V'	•	•	•	•
Intercept SIT	's'	'I'	•	•	•	•

Continued on next page

Table A-10. Call Dispositions for tic('D') and tic('O') — *Continued*

Meaning	TSM Level		Available On:			
	r.0	r.1	TR	T1	PRI	LST1
Ineffective other SIT	's'	'O'	•	•	•	•
Domestic other SIT	's'	'd'	•	•	•	•
International other SIT	's'	'o'	•	•	•	•
International no circuit SIT	's'	'c'	•	•	•	•
Foreign fail SIT	's'	'f'	•	•	•	•
Unknown SIT	's'	'U'	•	•	•	•
Touch tone entry detected	't'	'O'	•	•	•	

* By default, answer detection is disabled on T1 and PRI channels. However, it can be enabled using the **setcca** script instruction.

† The disposition of calls on PRI channels is based not only on Full CCA but also on information provided by the switch. When this particular disposition is provided by the switch, more specific information (the ISDN cause value) is available in register 1 (r.1). Refer to Table A-11 for the list of ISDN Cause Values. If this disposition is provided by Full CCA, register 1 always contains zero.

‡ For touch tone, any string with a character other than 1,2,3,4,5,6,7, 8,9,0,#,*,(,), or - is invalid. For dial pulse, any string with a character other than 1,2,3,4, 5,6,7,8,9,0,(,), or - is invalid. For T1 and PRI, the maximum dial string is 15 digits. For T/R 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.

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 SP).

- When V, P, 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 A-11 lists the ISDN cause values.
- When Register r.0 is set to 's', the Special Information Tone (SIT) is in r.1. The SIT types are listed in Table A-12.

ISDN Cause Values

Table A-11 lists the ISDN Cause Values returned in r.1 for a given call disposition returned in r.0 of the tic instruction when an outdial is used. For the call dispositions not listed in Table A-11, no additional information is available in r.1.

Table A-11. ISDN Cause Values

Call Disposition Value (r.0)	ISDN Value(r.1)	Meaning
Vacant code ('v')	1	Unassigned number
	22	Number changed
Provisioning or protocol error('p')	6	Channel Unacceptable
	18	No user response
	30	Status enquiry
	31	Normal unspecified
	50	Facility not subscribed
	52	Outgoing calls barred
	54	Incoming calls barred
	58	Bearer not available
	63	Service not available
	65	Bearer not implemented
	66	Channel not implemented
	69	Facility not implemented
	81	Invalid call reference
	82	Nonexistent channel
	88	Incompatible destination
	96	Info element missing
	97	Nonexistent message type
	98	Incompatible message
	100	Invalid info element
	102	Recovery on timer
	127	Internetworking unspecified
Disconnection detected ('D')	16	Normal clearing

Continued on next page

Table A-11. ISDN Cause Values — *Continued*

Call Disposition Value (r.0)	ISDN Value(r.1)	Meaning
Busy ('B')	17	User busy
Fast busy ('F')	2	No route
	21	Call rejected
	29	Facility rejected
	34	No circuit
	38	Network out-of-order
	41	Temporary failure
	42	Switching congestion
	43	Access info discarded
	44	Circuit not available
	45	Pre-empted
Illegal dial string (-3)	28	Invalid number

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.

Special Information Tones

The special information tones (SITs) provided in Table A-12 are returned to the script in Register r.1.

Table A-12. Special Information Tones

tic Return	
Value (r.1)	**Disposition**
R	Reorder, intraLATA
r	Reorder, interLATA
K	No circuit, intraLATA
k	No circuit, interLATA
V	Vacant code
I	Intercept
O	Ineffective other
D	Domestic other
o	International other
c	International no circuit
f	International foreign failure
U	Unknown SIT type

Example

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(im.1,im.10,im.-1)
nextcall:
dbase( .... )     /* get number to dial from DIP */

tic('O', r.3)    /* call number in register 3 */

jmp(r.0 == im.'N', noAns)     /* no answer after 10 rings */
jmp(r.0 == im.'B', busy)
jmp(r.0 == im.'F', retry)
jmp(r.0 == im.'A', answer)
jmp(r.0 == im.'s', SIT)
jmp(r.0 == im.-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 == im.'R', retry)
jmp(r.1 == im.'r', retry)
jmp(r.1 == im.'K', retry)
jmp(r.1 == im.'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)
```

Using tic with PRI

The supported **tic** instruction options for PRI are listed in Table A-13. These options are used in the same manner for PRI as for T1.

Table A-13. tic Options Supported for the PRI

Option	Function
a	Answer an incoming call
h	Disconnect (hangup) a call
o	Originate a call
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 A-14.

Table A-14. 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('O') and Disconnect Event Return Codes

The PRI implementation of the tic('O') (Originate) instruction provides additional return code information over the T1 and Analog interface implementations. Register *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 A-11 contains a list of ISDN cause values returned in register r.1.

All of the ISDN return codes returned in register r.1 are classified into several groups. These groups are shown in Table A-15 (labeled Return Values). A call disposition value gives a more general indication of the type of problem encountered. The disposition value is returned in register *r.0*.

Table A-15. tic ('O') Return 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
Provisioning or Protocol Error	CV_UNACCEPTCHAN (6) CV_NOUSER (18) CV_STATRESP (30) CV_NORMALUNSP (31) CV_FACNOTSUB (50) CV_OUTBARRED (52) CV_INBARRED (54) CV_BEARERNA (58) CV_SERVICENA (63) CV_BEARERNI (65) CV_CHANNELNI (66) CV_FACILITYNI (69) CV_INVALIDCALL (81) CV_NOCHANNEL (82) CV_BADDEST (88) CV_MISSINGIE (96) CV_BADMESS (97) CV_BADSTATE (98) CV_INVALIDIE (100) CV_TIMEOUTREC (102) CV_INTERWORKING (127)	Channel unacceptable No user response Status enquiry Normal, unspecified Facility not subscribed Outgoing calls barred Incoming calls barred Bearer not available Service not available Bearer not implemented Channel not implemented Facility not implemented Invalid call reference Nonexistent channel Incompatible destination Info element missing Nonexistent message type Incompatible message Invalid info element Recovery on timer Interworking, unspecified
Internal Protocol Error ('E')	CV_NULL (0) CV_FR (29) CV_TFAIL (41) CV_IDVFN (85) CV_TNDNE (91) CV_INVMSG(95) CV_NEIE (99) CV_PEU (111)	No cause value present Facility rejected Temporary resource failure on remote end Invalid digit value for number Transit network does not exist Invalid message Nonexistent IE Protocol error unspecified
Dialtone Detected ("D')	CV_NORMALCLR (16)	Normal clearing
Busy ("B')	CV_USERBUSY (17)	User busy

Continued on next page

Table A-15. tic ('O') Return Values for PRI — *Continued*

Call Disposition Value (r.0)	Return Value (r.1)	Meaning
Fast Busy ('F')	CV_NOROUTE (2) CV_CALLREJECT (21) CV_FACREJECT (29) CV_NOCIRCUIT (34) CV_NETWORKDOWN (38) CV_TEMPFAILURE (41) CV_SWITCHBUSY (42) CV_USERIETOSS (43) CV_CIRCUITNA (44) CV_CALLPREEMPTED (45)	No route Call rejected Facility rejected No circuit Network out of order Temporary failure Switching congestion Access info discarded Circuit not available Pre-empted
Answer Supervision ('P')		
Hardware Failure, Undetermined Reason (-1)		
Timeout - No Answer, Reason Not Provided		
Illegal Dial String (-3)	CV_INVALIDNUM (28)	Invalid Number

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 prevalent, -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"
```

Figure A-3 provides an example of how you might 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.

```
#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 == imm.’P’, Continue)/* Call answered, speak */
     /* Call was not answered */
     /* Did PRI indicate that the number does not exist? */
     jmp (r.1 == imm.CV_UNASSNUM, NumberUnassigned)
     /* Did PRI indicate that the number was incomplete? */
     jmp (r.1 == imm.CV_INVALIDNUM, NumberIncomplete)
     /* Did PRI indicate that the number has been changed? */
     jmp (r.1 == imm.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()
```

Figure A-3. PRI Application Using the tic Instruction

tnum

Name

This script instruction speaks a number with natural speech.

Synopsis

tnum(*type.src [,'inflection']*)

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 **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 standard 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 **getdig** 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:
  getdig(0,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(im.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 *Intuity CONVERSANT VIS V5.0 Command Reference,* 585-310-230.

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(im.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(im.1000, im."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 AT&T stock for the
price of")
tnum(int.SELL_PRICE)
   ........................   ......................
```

See Also

tflush

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 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 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 **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, **getdig** would require a minimum of seven digits.

The **ttdelim** instruction works with the **getdig** and **tttime** instructions. For example, after requesting 5 digits with a 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 **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 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 getdig instruction. For example, if 8 digits plus a touch-tone terminator are expected, getdig must look for 8 digits plus the length of the touch-tone terminator.

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 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 **getdig** instruction. If the *firstdig* time is exceeded, r.0 is set to 0 and the **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 **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(im.4,im.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*)

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 to play a short "beep" just before voice coding starts.

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, im.ADPCM32)
or(int.CODETYPE, im.NO_AGC)
vc('b', im.60, int.CODETYPE)
```

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. 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,im.100)
vc (’p’,int.TIME,im.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,im.SBC16)
vc(’b’,im.120,short.RATE)
```

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(im.3,im.3)
```

Voice System C-Library Functions

B

What's in This Appendix

This appendix contains summaries of the C-library functions discussed earlier in this book. This chapter is divided into two sections according to the library in which the functions reside.

The functions are listed in alphabetical order under the library which they are located. Each function appears on a separate page and, for each function, the following is provided:

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

These library pages should be used to locate detailed information about each function.

Table B-1 lists the functions in alphabetical order and the library in which the function resides. If an Intuity Response Application Programming Interface (IRAPI) equivalent exists, it is listed in column 2 of the table.

NOTE:
The libspp.a functions have been replaced by IRAPI functions. All new data interface processes (DIPs) should be written in terms of the IRAPI. Refer to the *Intuity CONVERSANT VIS V5.0 IRAPI Programming Guide*, 585-310-226, for additional information.

Table B-1. C-Library Function Locations

Function	IRAPI Equivalent	Library
arrays		libprism.a
BitMasks		libprism.a
CharBuffer		libprism.a
clock		libprism.a
db_init	irRegister	libspp.a
db_pr	irTrace	libspp.a
db_put	irTrace	libspp.a
et_send		libspp.a
expandLog		liblog.a
ipc		libprism.a
logDstPri		liblog.a
logMsg		liblog.a
match		libprism.a
mesgrcv	irWcheck	libspp.a
mesgsnd	irPostEvent	libspp.a
options		libprism.a
parseln		libprism.a
readLine		libprism.a
regex		libprism.a
startup	irRegister	libspp.a
strmatch		libprism.a
threshold		libalerter.a
timeIncr		libprism.a
tmtotime		libprism.a
usage		libprism.a
VSerror		libspp.a
VSstartup	irRegister	libspp.a
VStoname		libspp.a
VStoQkey	irGetQkey	libspp.a

libspp.a 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 VIS. 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 (refer to *Intuity CONVERSANT VIS V5.0 Command Reference*, 585-350-230). 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 appeas 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 *UNIX System V/386 Release 3.1 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

et_send

Name

et_send — Send error message to ET

NOTE:
This function is obsolete in the Version 5.0 logger/alerter environment.

Synopsis

```
#include "spp.h"

int    et_send (chan,msg_id,e_arg0,e_arg1,e_arg2,e_arg3,e_strarg)
int    chan;          /* channel in lower 2 bytes, board in upper 2 bytes*/
int    msg_id;        /* value of mnemonic #defined for error type */
int    e_arg0;        /* integer argument */
int    e_arg1;        /* integer argument */
int    e_arg2;        /* integer argument */
int    e_arg3;        /* integer argument */
char   *e_strarg;     /* string argument */
```

Description

et_send is used to notify ET of an error in the calling process. It sends an IPC message to ET with the specified arguments. Errors are identified by their error ids (*msg_id*) and should be unique across all errors known to ET. See the files under /att/msgipc/etmsgs for the errors currently known to ET.

ET generates the text description of the error by applying the arguments *e_arg0* through *e_arg3* and *e_strarg* to the rule associated with the given error (msg_id). The rule also tells ET what action to take; this usually translates to logging the error and text description. The error log can be displayed through the Event Log Report menu via the cvis_menu command.

Set the channel number to -1 if you are not sure of the channel and board number. This is important because if the channel and board number are incorrect, ET will reject the message. The advantage to specifying the channel and board number, if you know them, if that the message will identify the board and channel on which the error occurred. Set any of the integer arguments (*e_arg0* through *e_arg3*) to -1 and the *e_strarg* to the empty string (" ") if they are not used.

Example

Assume that *msg_id*, DIP_FILE_ERR is known to ET and it substitutes the arguments to et_send using the following format string:

```
"database DIP:channel %chan: %st error (errno=%arg0)"
```

The actual chan, e_strarg, and e_arg0 arguments passed to et_send replace *%chan*, *%st*, and *%arg0*, respectively. The above format string would be used as part of the rule associated with DIP_FILE_ERR. The following example displays the code fragment for sending the DIP_FILE_ERR:

```
extern int errno: /* (see intro(2)). */
int  fd;
int  chan;
char but[32]
/* Assume working on channel 10 on board 1 but usually
* information would not be hardcoded as in this example.
* Instead, the channel number is extracted from an IPC
* message that is usually sent by a TSM script.
*/
chan = 10|(1>>16);
fd = open("customer"); /* customer records are kept here */
if (fd < 0) {
  /* Assuming that errno is set to 13 (access denied),
  * the following et_send message will be logged as
  * "databaseDip: channel 10: OPEN error (errno=13)"
  */
  et_send(chan, DIP_FILE_ERROR, errno, -1, -1, -1, "OPEN");
  /* abort further processing with this file */
} else { /* open worked */
  noBytes = read(fd, buf, 32);
  if (noBytes < 0) {
    /* Assuming that errno is set to 4 (interrupted call).
    * The following et_send message will be logged as
    * "databaseDIP: channel 10: READ error (errno=4)"
    */
    et_send(chan, DIP_FILE_ERROR, errno, -1, -1, -1, "READ")"
    /* abort further processing with this file */
  }
}
```

Diagnostics

et_send does not return any indication of success or failure. If it fails to send the IPC message, et_send prints out a diagnostic on standard output (stdout).

Warning

et_send will truncate *e_strarg* to 64 (ET_MAXSTR-1) characters if necessary and will not print out a diagnostic when this happens.

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
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 */
```

Description

Mesgrcv is used by the voice system to read IPC messages off their message queues. It is the front-end to the UNIX system call, msgrcv. 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 & msgflag) is true, the message will truncated to fit in the buffer (see **msgop(2)** in the *UNIX SVR4.2 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 & *msgflag* is true.

Mesgrcv creates the IPC message queue for queue key *morig* if necessary.

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) (*UNIX SVR4.2 Command Reference*)
msgsnd(2(2)) (*UNIX SVR4.2 Command Reference*)

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 UNIX system call msgsnd. *Msgflag* is sent directly to the UNIX 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) (*UNIX SVR4.2 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 pre-defined 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 the discussion of VSstartup later in this section.

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 VIS 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 UNIX system errors and the appropriate text in the UNIX error table (sys_errlist[]) is returned. Error ids defined in VS.h and UNIX 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 VIS. 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 pre-defined 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_ELEN	Length of *procName* is out of range
VS_ERESV	*procName* is reserved for hardcoded processes
VS_ENOPRT	Non-printable character found in *procName*
VS_ENUM	*Instance* is negative or out of range
VS_BADPROC	Another process with the same *procName* and *instance* is running already
VS_ENOFREE	No BB slots available for posting process (see troubleshooting information in Chapter 4, "Data Interface Process" for more information)
VS_ESHMAT	Can 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.

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 Process" for more information)

Warning

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 non-existent procNames.

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 com-
puted. */
    int th_flags ;

#define TH_EDGE_TRIGGERED  0x0000   /* Respond when level exceeded
*/
#define TH_LEVEL_TRIGGERED 0x0001/* Respond whenever above level */
#define TH_NO_REPORT       0x0002   /* 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

It is strongly urged that you use the fast malloc () routines found in -lmalloc due to the fact that a great deal of allocating and deallocating is performed by these routines.

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. s etExpandFmt 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 non-printing characters be made visible. getVisible returns 0 if control and non-printing 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
LOGCONTPREFIX	The 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 UNIX system whenever a system call fails.

See Also

arrays
expandLog
fixLogFile
LComp
log
logDstPri

libprism.a Functions

The following information is provided in this appendix for completeness. These functions arenot required for proper operation of the TAS script or DIP, but some information contained here may be useful.

arrays

Name

arrayOpen, arrayPut, arrayVPut, arrayDel, arrayClose, arrayDone, arrayDetach, arrayTransfer, arrayDesc, arrayPtr, arrayCnt, arrayChgIncr — Functions for managing dynamic arrays

Synopsis

```
#include <sys/types.h>
#include "prismdefs.h"
#include "arrays.h"

int arrayOpen(type,size)   /* Open dynamic array */
int type ;    /* primary or secondary */
unsigned size ;            /* size of objects */

void *arrayPut(ardes,ptr)  /* Constant length item array put */
int ardes ;  /* Dynamic array descriptor */
caddr_t ptr ;              /* Pointer to data to be stored */

void *arrayVPut(ardes,ptr,len)/* Variable length item array put */
int ardes ;  /* Dynamic array descriptor */
caddr_t ptr ;              /* Pointer to data to be stored */
unsigned len ;             /* Length of variable length item */

int arrayDel(ardes,ptr)    /* Delete item from array */
int ardes ;  /* Dynamic array descriptor */
caddr_t ptr ;              /* Pointer to item to be deleted */

int arrayClose(ardes)      /* Close & release array */
int ardes ;  /* Dynamic array descriptor */

void **arrayDone(ardes)    /* Close array, BUT DO NOT release. */
int ardes ;  /* Dynamic array descriptor */

void **arrayDetach(ardes)  /* Detach array. Leave array open. */
int ardes ;  /* Dynamic array descriptor */

int arrayTransfer(ardes)   /* Transfer array contents. */
int ardes ;  /* Dynamic array descriptor */

ARRAY *arrayDesc(ardes)    /* Get array descriptor pointer */
int ardes ;  /* Dynamic array descriptor */
```

```
void **arrayPtr(ardes)       /* Get array data pointer */
int ardes ;     /* Dynamic array descriptor */

int arrayCnt (ardes)         /* Get count of items in array */
int ardes ;     /* Dynamic array descriptor */

unsigned arrayChgIncr(ardes,incr)
int ardes ;     /* Dynamic array descriptor */
unsigned incr ;              /* New increment size for array */
```

Description

These functions can be used by programs to store data in dynamically allocated arrays. Arrays come in two types, *DA_PRIMARY* and *DA_SECONDARY*. A primary array is one in which the data is stored directly in the array. A secondary array is one in which the array contains a series of pointers to areas that contain the data. If you needed a dynamic array of longs, you would do:

```
"ardes = arrayOpen(DA_PRIMARY,sizeof(long)) ;"
```

If you wanted to build arrays of *field_item* structures such as those used in TABS, you would want a secondary array, where the array contained a series of pointers to the actual *field_item* structures as shwon in the following:

```
"ardes = arrayOpen(DA_SECONDARY,sizeof(struct field_item)) ;"
```

The arrays are dynamic in that as more items are stored in the array, the bigger the array gets. New space is allocated any time the current array area runs out of room. These functions are particularly useful when programmers want to avoid setting a predefined internal limit for data storage. Functions returning (void *) data type should have their return values cast to the appropriate data type pointer (see the Example section for this function).

By convention, DA_SECONDARY type arrays always have a NULL pointer at one past the end of the array. This can be used to indicate that the end of the array has been reached when scanning it pointer by pointer and it makes an array space of (char*) pointers, as created with arrayVPut ( ), appropriate as a target for freeStrArray ( ). (See freeStrArray (3X)) if it has been detached from its dynamic array with arrayDone ( ) or arrayDetach ( ).

arrayOpen ( ) opens a dynamic array. It returns an array descriptor which is used with the other functions to reference the opened array. *Type* specifies whether the array directly contains the data items, that is, is a primary array, or contains pointers to the data items, that is, is a secondary array. *Size* is the number of bytes of each data item to be stored. *Size* may be 0 if the type of the array is secondary. This implies that the array is a variable-length-item array and data can only be placed in the array with the arrayVPut routine. arrayOpen ( ) returns NULL upon error.

arrayPut ( ) is used to add a data item to a dynamic array. *ardes* is an array descriptor obtained from arrayOpen ( ). *ptr* is a pointer to the data to be stored. arrayPut ( ) returns a pointer to the stored data if it succeeds or NULL upon error. arrayPut ( ) cannot be used to store data in a variable-length-item array. If *ptr* is NULL, a null object is inserted in the array at this point. This means that the pointer returned, points to an object that is all zeros.

arrayVPut ( ) is used to add a variable length data item to a dynamic array. *ardes* is an array descriptor obtained from arrayOpen ( ). The array must be a secondary type array specifically opened with an object size of 0. *ptr* is a pointer to the data to be stored. *len* is the length of the item to be stored in the array. It must include NULL terminating bytes in cases where the items are NULL terminated strings. arrayVPut ( ) returns a pointer to the stored data if it succeeds or NULL upon error. If *ptr* is NULL, a null object is inserted in the array at this point. This means that the pointer returned, points to an object that is all zeros.

arrayDel ( ) is used to delete a data item from a dynamic array. *ardes* is an array descriptor obtained from arrayOpen ( ). *ptr* is a pointer to the array data to be deleted. It should be a value returned by an arrayPut ( ). If successful, this arrayDel ( ) removes the data and moves all subsequent pointers in the array up. arrayDel ( ) returns SUCCESS or FAILURE.

arrayClose ( ) frees up all data items and their pointers in the dynamic array associated with *ardes*. arrayClose ( ) returns SUCCESS or FAILURE.

arrayDone ( ) is similar to arrayClose ( ) in that it closes the array descriptor, *ardes*, BUT it does not release the array space itself, instead returning a pointer to the array. arrayDone is used in cases where there is a growing phase of the array and then a usage phase where the array does not grow. For example, in an application that was reading in arbitrary arrays of information initially, followed by use of those arrays, the application might not want to tie up an array descriptor indefinitely once the array was read. arrayDone basically disassociates the array from the array descriptor, closes the array descriptor, and returns a pointer to the array for the application to work with. Once an arrayDone is performed, the dynamic array module no longer maintains any information about the array and cannot be used to add or delete elements from the array. It is then the responsibility of the application to maintain the array and/or release it.

arrayDetach ( ) is similar to arrayDone ( ) in that it returns a pointer to the array space associated with the dynamic array specified by *ardes*. As with arrayDone ( ), this allocated array is no longer associated with a dynamic array and is the responsibility of the caller to manage. The difference between arrayDetach ( ) and arrayDone ( ) is that arrayDetach ( ) leaves the dynamic array OPEN with its count of items and current array size set back to zero. This is useful if one routine wants to take over management of the array space created by another routine without causing the array descriptor to become invalid. This would be necessary if the originating routine had the array descriptor stored in a static, which was therefore inaccessible to any other routine to set to zero (indicating the array was not open.) Examples of this are the parselnDA ( ) and parselnEnhancedDA ( ),

both of which maintain their own internal and open dynamic arrays for parsing. The calling routine is expected to examine the array, but not close it. These routines empty their arrays prior to each new parsing. The use of arrayDetach ( ) allows a user of these routines to take control of the parsed information.

arrayTransfer ( ) opens a new dynamic array, transfers the array associated with the original array descriptor, including the type, the increment size, and all other details, to the new dynamic array, and disassociates the array space from the original array, setting its count of items back to zero. arrayTransfer ( ) can be used when two routines wish to cooperate in the management of a dynamic array, but want to maintain control of their own array descriptor. For example, if you want to use parselnDA ( ) or parselnEnhancedDA ( ) to create a dynamic array of words, but then want to perform further operations on the dynamic array, arrayTransfer ( ) would allow you to create your own dynamic array descriptor that contained the information created by one of the *parseln* ( )* routines. If you did not use arrayTransfer ( ), the information in the array would be freed at the next call to the parseln* ( ) routines.

arrayDesc ( ) is a macro which returns the pointer to the dynamic array descriptor structure (type: *"ARRAY *")*.

arrayPtr ( ) is a macro which returns the pointer to the dynamic array itself.

arrayCnt ( ) is a macro which returns the count of the number of items currently stored in the dynamic array.

arrayChgIncr ( ) allows the user of an array to specify the granularity of the expansions of the array. The default array size is 10. Whenever growth is required, 10 more elements are added to the current array during reallocation. If a very large array was going to be used, specifying a larger granularity would increase efficiency, since reallocating space tends to be costly in terms of CPU time. arrayChgIncr returns the previous granularity and sets the granularity to *incr.*

Example

The following example illustrates the use of the above routines.

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

#include "prismdefs.h"
#include "arrays.h"

main( )
  {
    int fiArdes,timeArdes ;
    struct field_item field ;
    FILE *fp ;
    struct field_item **fipp ;
    extern long time( ) ;
    extern char *ctime( ) ;
```

```
    /* Open two dynamic arrays, one for a secondary array of
    * of field_item structures like TABS uses,
    * and one for an array of times.
    */

    fiArdes = arrayOpen(DA_SECONDARY,sizeof(struct field_item)) ;
    timeArdes = arrayOpen(DA_PRIMARY,sizeof(time_t)) ;

    fp = fopen("testForm","r") ;

    while (readForm(fp,&field) == TRUE)
      {

          fipp = (struct field_item **)arrayPut(fiArdes,&field) ;
          /* ... */
      }

    for (;;)
      {
          /* Display form */

          Get_SI_List(arrayPtr(fiArdes),array-
Cnt(fiArdes),0,FALSE,FALSE) ;

          /* Save a time stamp each time through the form */

          (void)arrayPut(timeArdes,time((long*)NULL)) ;

          /* ... */
      }
    /* Print out the times and delete them as you go */

    for (i=0; arrayCnt(timeArdes) > 0 ;i++)
      {
          fprintf(stdout,"%3d Time: %s",i,ctime(arrayPtr(time-
Ardes))) ;
          /* This is not a particularly good example since the
           * deletion causes all the items above to be moved. A
           * better example would be where you wanted to remove
           *  randon items from in the interior of the array.
           */

          arrayDel(timeArdes,arrayPtr(timeArdes)) ;
    }
    /* Free all space associated with this array */

    arrayClose(fiArdes) ;
    arrayClose(timeArdes) ;

    exit(0);
}
```

See Also

parseln

bitMasks

Name

mkBitMask, setBitMask, clrBitMask, complimentBitMask, tstBitMask, nextBitMask, mkCopyBitMask, orBitMasks, andBitMasks, andComplimentBitMasks, xorBitMasks, freeBitMask, setRangeBitMask, clrRangeBitMask, complimentRangeBitMask, tstRangeBitMask, zeroBitMask — Routines to create and manipulate dynamic extensible bit masks.

Synopsis

```
#include prismdefs.h
#include bitMasks.h

struct BitMask *mkBitMask ( sz,incr )
int sz ;              * Initial size of bit mask, normally 0 */
int incr ;             /* Amount to increment bit mask by when
expanding */

struct BitMask *setBitMask ( bmp,bitPos )
struct BitMask *bmp ;
int bitPos ;

struct BitMask *clrBitMask ( bmp,bitPos )
struct BitMask *bmp ;
int bitPos ;

struct BitMask *complimentBitMask ( bmp,bitPos )
struct BitMask *bmp ;
int bitPos ;

int tstBitMask ( bmp,bitPos )
struct BitMask *bmp ;
int bitPos ;

int nextBitMas” ( bmp,startPos,clrFlag )
struct BitMask *bmp ;
int startPos ;
int clrFlag ;         /* If TRUE, clear each bit as found */

struct BitMask *mkCopyBitMask ( srcbmp )
struct BitMask *bmp ;

struct BitMask *orBitMask ( bmp1,bmp2 )
struct BitMask *bmp1 ;
struct BitMask *bmp2 ;

struct BitMask *andBitMask ( bmp1,bmp2 )
struct BitMask *bmp1 ;
struct BitMask *bmp2 ;
```

```
struct BitMask *andComplimentBitMask ( bmp1,bmp2 )
struct BitMask *bmp1 ;
struct BitMask *bmp2 ;

struct BitMask *xorBitMask ( bmp1,bmp2 )
struct BitMask *bmp1 ;
struct BitMask *bmp2 ;

struct BitMask *freeBitMask ( bmp )
struct BitMask *bmp ;

struct BitMask *setRangeBitMask ( bmp,low,high )
struct BitMask *bmp ;
int low ;
int high ;

struct BitMask *clrRangeBitMask ( bmp,low,high )
struct BitMask *bmp ;
int low ;
int high ;

struct BitMask *complimentRangeBitMask ( bmp,low,high )
struct BitMask *bmp ;
int low ;
int high ;

int tstRangeBitMask ( bmp,low,high )
struct BitMask *bmp ;
int low ;
int high ;

void zeroBitMask ( bmp )
struct BitMask *bmp ;
```

Description

A BitMask structure is a dynamic object containing an indefinitely extensible bit mask. BitMask structures are created by any operation that attempts to add or delete bits from a bit mask, that is, setBitMask ( ), clrBitMask ( ), complimentBitMask ( ), mkCopyBitMask ( ), orBitMask ( ), andBitMask ( ), andComplimentBitMask ( ), xorBitMask ( ), setRangeBitMasks ( ), clrRangeBitMasks ( ), and complimentRangeBitMasks ( ). In addition they can be created directly with mkBitMask ( ).

Using mkBitMask ( ) to create a BitMask, *sz* is the initial size of the bit mask. It is perfectly reasonable to set this to 0. If so, no space will be allocated for the bit mask until an attempt is made to set a bit in the mask. *incr* is the size of each expansion of the bit mask. If 0 is specified, the default, which is 4, is used, meaning that each expansion of the bit mask will increase its size by four words.

setBitMask ( ), clrBitMask ( ), and complimentBitMask ( ) respectively set, clear, and compliment the bit specified by bitPos in the bit mask. If bmp is NULL, the bit mask is first created before the operation is performed.

tstBitMask ( ) returns TRUE if the bit specified by bitPos is set otherwise it returns FALSE. nextBitMask ( ) finds the next set bit in the bit mask and returns its position. It starts its search at startPos. If *clrFlag* is set, it also clears each bit that it finds. FAILURE is returned if there are no more bits set between startPos and the top of the bit mask.

mkCopyBitMask ( ) creates a copy of the specified bit mask.

orBitMasks ( ), andBitMask ( ), andComplimentBitMasks ( ), and xorBitMasks ( ) perform the specified operations between two bit masks. In all cases *bmp1* is the target bit mask and *bmp2* is the bit mask manipulated to affect *bmp1*. For example, with andComplimentBitMasks ( ), *bmp2* is complimented and its bits then ANDed with the bits in *bmp1* thereby producing a new *bmp1* missing those bits set in bmp2.

freeBitMask ( ) frees the resources associated with the bit mask *bmp*.

setRangeBitMask ( ) set all the bits from *low* through *high* bit positions. The bit mask is created and extended as necessary to perform the operation.

clrRangeBitMask ( ) clears all the bits from *low* through *high* bit positions. The bit mask is created and extended as necessary to perform the operation.

complimentRangeBitMask ( ) compliments all the bits from *low* through *high* bit positions. The bit mask is created and extended as necessary to perform the operation.

tstRangeBitMask ( ) tests all the bits from *low* through *high* bit positions. TRUE is returned if any of the bits are set. FALSE is returned if the bit mask has not been created yet or does not span the specific range of bits.

zeroBitMask ( ) zeros all the bits in the specified bit mask by releasingany allocated space associated with the bit mask and setting its length to zero.

It is not necessary that the bit masks being manipulated be of the same size when they are logically combined. It is always assumed that the shorter bit mask contains zero bits in all non-existant parts and so the affect on the larger bit mask is based on the operation being performed against 0 bits.

Examples

Create two bit masks for each person in an arbitrary database. One bit mask is those people who lives in Texas. The other bit mask is those people who are unemployed. AND them together to create a third bit mask containing the unemployed Texans:

```
int bitPos ;
struct BitMask *txBmp,*unemployedBmp,noJobsTxBmp ;
struct Record *rp ;
extern struct Record readRecord() ;
for (bitPos=0,noJobsTxBmp=txBmp=unemployedBmp=(struct BitMask
*)NULL
  ; rp = readRecord() ;bitPos++)
  {
    if (strcmp(rp->state,"Texas") == 0)
      txBmp = setBitMask(bmp,bitPos) ;
    if (rp->job == NONE)
      unemployedBmp = setBitMask(unemployedBmp,bitPos) ;
  }
noJobsTxBmp = mkCopyBitMask(txBmp) ;/* Make copy so txBmp can be
saved.*/
noJobsTxBmp = andBitMasks(noJobsTxBmp,unemployedBmp) ;
```

Notice that all three bit masks start out as NULL pointers. If no records were detected for either one case or the other, the code still works properly with NULL bit mask pointers.

CharBuffer

Name

mkCharBuffer, freeCharBuffer, detachCharBuffer, putCharBuffer, fputCharBuffer, putStrCharBuffer, removeLastCharBufferChar, resetCharBuffer, fullnessOfCharBuffer, sizeofCharBuffer, nullTerminateCharBuffer, fmtStr, vfmtStr, fmtCharBuf, vfmtCharBuf, maxFormatLength, vmaxFormatLength — Automatic resizing character buffer routines

Synopsis

```
#include <stdio.h>
#include <varargs.h>
#include prismdefs.h
#include  charBuffer.h

struct CharBuffer *mkCharBuffer ( incr ) /* Create CharBuffer */
int incr ;

void freeCharBuffer ( bp )
struct CharBuffer *bp ;

char *detachCharBuffer ( bp )           /* Return only buffer */
struct CharBuffer *bp ;

int putCharBuffer ( bp,chr )            /* Put character in buffer - MACRO
*/
struct CharBuffer *bp ;
char chr ;

int fputCharBuffer ( bp,chr )           /*Put character in buffer  */
struct CharBuffer *bp ;
char chr ;

char *expandCharBuffer ( bp )           /* Expand buffer by 1 increment */
struct CharBuffer *bp ;

char *putStrCharBuffer ( bp,str )       /* Put string in buffer */
struct CharBuffer *bp ;
char *str ;

int removeLastCharBufferChar ( bp )     /* Remove & return last char */
struct CharBuffer *bp ;

void resetCharBuffer ( bp )             /* Empty buffer */
struct CharBuffer *bp ;
```

```
int fullnessOfCharBuffer ( bp )     /* Return # of chars in buffer */
struct CharBuffer *bp ;

void sizeofCharBuffer ( bp )        /* Return current buffer size */
struct CharBuffer *bp ;

char *charBufferAdr ( bp )          /* Return current ptr to buffer */
struct CharBuffer *bp ;

char *nullTerminateCharBuffer ( bp ) /* Insure '\0' char at end */
struct CharBuffer *bp ;

char *fmtStr ( fmt[,arg...] )       /* Create formatted string */
char *fmt ;

char *vfmtStr ( fmt,argp )          /* varargs version of  fmtStr  */
char *fmt ;
va_list *argp ;

int fmtCharBuffer ( bp,fmt[,arg...) /* Add formatted str to buffer */
struct CharBuffer *bp ;
char *fmt ;

int vfmtCharBuffer ( bp,fmt,argp ) /* varargs version of  fmtCharBuffer  */
struct CharBuffer *bp ;
char *fmt ;
va_list *argp ;

int maxFormatLength ( fmt[,arg...] ) /* Return max format buffer size */
char *fmt ;

int vmaxFormatLength ( fmt,argp ) /* varargs version of  maxFormatLength
*/
char *fmt ;
va_list *argp ;
```

Description

These routines manipulate character buffers that are automatically expanded when they are found not to be large enough. They are adjuncts to the other automatic buffering routines found in PRISM TOOLS. See readLine (3x), readFile (3x), and arrays (3x).

mkCharBuffer ( ) creates the control structure to manage a character buffer. incr specifies the size of each increment to the character buffer when it needs expansion. The default, if incr is less than or equal to 0, is 0x80 bytes.

The control structure should be released via either the **freeCharBuffer ( )** routine, which releases both the control structure and any data in the character buffer, or

the **detachCharBuffer ( )** routine, which releases only the control structure and returns the completed character buffer as its return value. When using detach**CharBuffer ( )** it is the responsibility of the user to *free* ( ) the character buffer if that should ever be appropriate.

putCharBuffer ( ) adds a single character to the specified character buffer, expanding the buffer if necessary. **putCharBuffer ( )** is a macro. It returns the character stored or FAILURE if the buffer fails to expand when it should. It does not call the character buffering routines unless the buffer needs enlarging.

fputCharBuffer ( ) performs the same job as **putCharBuffer ( )** but it is a function and so incurs the expense of a subroutine call each time it is called. It returns either the character stored or FAILURE if bp is not initialized or the buffer fails to expand when it should.

WARNING:
Be aware that a CharBuffer *filled via either the* putCharBuffer ( ) *or* fputCharBuffer ( ) *routines will not contain a '\0' character at the end of the buffer unless you explicitly store a '\0' character at the end or use* nullTerminateCharBuffer ( ) *to insure that there is a '\0' character at the end of the buffer. Without this termination character, standard UNIX string operations will produce unpredictable results.*

putStrCharBuffer ( ) adds an entire null terminated ASCII string to the specified character buffer. **putStrCharBuffer ( )** insures that the CharBuffer has a '\0' character after the last real character in the buffer. If *str* is NULL or points the the empty string, the only action is to insure this null termination. The '\0' character is not actually part of the buffer and will be overwritten by the next **putCharBuffer ( )** operation.

removeLastCharBufferChar ( ) extracts the most recently added character from the specified buffer and removes it from the buffer. If the buffer is empty, **EOF** is returned. A side effect is that that buffer is guaranteed to be NULL terminated after a call if there was something to remove, since the removed character is converted to a '\0' character.

expandCharBuffer ( ) is normally only called by macros or internally by the character buffer routines. It causes the character buffer to be expanded by one increment of size (default 0x80 bytes) each time it is called. It returns the new pointer to the character buffer.

resetCharBuffer ( ) resets the pointer into **bp** so that future information is stored at the beginning of the CharBuffer. This effectively empties the buffer. **fullnessOfCharBuffer ( )** returns the number of characters currently stored in the **CharBuffer**. This count DOES NOT include the trailing NULL character generated by **nullTerminateCharBuffer ( )** or by any of the string appending

operations. **sizeofCharBuffer ( )** returns the current size of the CharBuffer, (that is, how many characters it can hold before it will be enlarged again). **charBufferAdr ( )** returns a pointer to the current buffer of characters associated with bp.

nullTerminateCharBuffer ( ) is a macro call to putStrCharBuffer that insures that the specified buffer has a '\0' character after the last valid character in the buffer so that it can be manipulated by the standard ASCII string routines.

fmtStr ( ) is a replacement for **sprintf ( )** in the standard I/O library. It returns a pointer to string containing the information specified by the format fmt and any required arguments. **vfmtStr ( )** is the varargs version, which takes a pointer to the required arguments rather than the arguments themselves. It replaces **vsprintf ( )**. Both **fmtStr ( )** and **vfmtStr ( )** use a single CharBuffer structure for all their formatting activities. Successive calls overwrite the previous values. If previous information must be retained, a copy must be made. (Use **mkcopy ( )**.)

fmtCharBuffer ( ) and **vfmtCharBuffer ( )** perform formatting activities directly into a CharBuffer structure pointed to by bp. The formatted information is concatenated to any information already in the **CharBuffer( )**. **fmtCharBuffer ( )** takes its arguments directly. **vfmtCharBuffer ( )** takes its arguments indirectly via *argp*. Both routines return the number of characters added to the **CharBuffer**.

maxFormatLength ( ) and vmaxFormatLength ( ) return the approximate length required to format the specified *fmt* format string using the supplied arguments. maxFormatLength ( ) takes its arguments directly, like printf ( ). vmaxFormatLength ( ) takes its arguments indirectly, via *argp*, like vprintf ( ). The length returned is guaranteed to be greater than or equal to the required length of the formatted material plus 1 addition byte for the '\0' character at the end.

Diagnostics

fputCharBuffer returns the character stored as an integer or FAILURE if the buffer pointer was not initialized or if the buffer failed to expand when needed. Both **putStrCharBuffer ( )**, and **expandCharBuffer ( )** return a pointer to the current buffer containing the characters stored to this point. **detachCharBuffer ( )** returns the pointer to the completed buffer. mkCharBuffer ( ) returns (struct CharBuffer *) NULL if it is unable to allocated space.

Example

The following example copies a line of input from the user into a buffer and then insures that it is NULL terminated and prints it.

```
#include  prismdefs.h
#include  readInfo.h
#include  charBuffer.h

struct CharBuffer *getInputAndEcho(fp)
FILE *fp ;
```

```
 {
  struct CharBuffer *cbp ;
  int chr ;

  cbp = mkCharBuffer(0) ;
  while ((chr = getc(fp)) != EOF && chr != '\n')
    putCharBuffer(cbp,chr) ;
  putCharBuffer(cbp,'\n') ;
  nullTerminateCharBuffer(cbp) ; /* Terminate ASCII string */
  fputs(charBufferAdr(cbp),stdout) ;/* Print contents */
  return(cbp) ;
}
```

The following routine reads a file and stores its entire contents as a single character buffer and returns the buffer. It is up to the caller to free the buffer space at some later time.

```
char *getFile(fp)
FILE *fp ;
  {
    struct CharBuffer *cbp ;
    char *ptr ;

    cbp = mkCharBuffer(0) ;
    while ((ptr = readLine(fp)) != EOF)
      (void)putStrCharBuffer(cbp,ptr) ;
    return(detachCharBuffer(cbp)) ;
  }
```

The following example takes a series of format strings and associated arguments and formats them into a character buffer and returns the composite buffer.

```
struct FormatItem
  {
    char *fmt ;
    va_list *argp ; /* Pointer to args for format */
  } ;

char *formatter(fip)

struct FormatItem *fip ;
  {
    struct CharBuffer *cbp ;

    if (fip == (struct FormatItem *)NULL)
      return((char*)NULL) ;

    cbp = mkCharBuffer(0) ;
    for (; fip->fmt != (char*)NULL ;fip++)
      (void)vfmtCharBuffer(cbp,fip->fmt,fip->argp) ;
    return(detachCharBuffer(cbp)) ;
  }
```

Caveats

Be aware that resetCharBuffer ( ) does not destroy the current contents of the buffer. It only resets the pointer back to the beginning. The next operation that puts something into the buffer while overlay (and thereby destroy) the previous contents.

Also be aware that the putCharBuffer ( ) operations do not insure NULL termination of the buffers contents. If you plan to perform any ASCII string operations on a buffer, be sure that either a nullTerminateCharBuffer ( ) operation is performed as the last operation or that one of the string concantenating operations, putStrCharBuffer ( ), fmtCharBuffer ( ), or vfmtCharBuffer ( ), is the last operation performed on the buffer. These operations guarantee that the buffer is NULL terminated.

See Also

readLine
readFile
arrays

clock

Name

clkinit, timeout, killtout, untimeout, sleep, setflag, suspendTimeouts, unsuspendTimeouts — Routines to manage a queue of timeout requests similar to the clock queues in UNIX

Synopsis

```
#include <sys/types.h>
#include clock.h

int clkinit ( ncallouts )
int ncallouts ;

int timeout ( func,arg,ticks )
int (*func)() ;
caddr_t arg ;
long ticks ;

long killtout ( func,arg )
int (*func)() ;
caddr_t arg ;

long untimeout ( id )
int id ;

unsigned sleep ( ticks )
unsigned int ticks ;

setflag ( flag )
int *flag ;

suspendTimeouts ()

unsuspendTimeouts ()
```

Description

clkinit ( ) sets up the callout queues. *ncallouts* is the size of the queues. If clkinit ( ) has not been called at the time the first **timeout ( )** call is made, the queues will be initialized at *DFL_CALLOUTS*, which is currently 20. **clkinit ( )** can also be used to resize the existing queues if necessary.

timeout ( ) sets up a request on the callout queues that the specified *func* be called in *ticks* seconds with *arg* as its single argument. If it is desired to just be interrupted at as certain time in the future without having a routine called, for

example, you want to attempt an **open ( )** for so many seconds and then quit if it does not succeed, timeout ( ) may be called with func set to **NOACTION**. **timeout ( )** returns FAILURE if there is not room for another timeout request. When it succeeds it returns a unique ID number than can be used to kill the timeout request via untimeout ( ).

killtout () removes a specific unexpired request from the callout queues. *func* and *arg* must match those specified in the **timeout ()** requeust. If two requests for the same function with the same argument were on the queues, only the first one would be removed. **killtout ()** returns the amount of time left on the request or FAILURE if it cannot find the request.

untimeout ( ) performs that same job as **killtout ( )** except that it uses the unique ID returned by **timeout ( )** to identify the timeout structure of interest. Its advantage is that if there is more than one request with the same *func* and *arg* , you can still specifically remove the desired request. **killtout ( )** always removes the first matching request.

sleep ( ) provides the same function as the UNIX C library sleep, but is compatible with timeout 's use of the *SIGALRM* signal. Processing is suspended for *ticks* seconds.

setflag ( ) is provided as a function to be called by **timeout ( )** to set a flag to TRUE when a timer expires. It is set up by the following call:

"timeout(setflag,&flag,ticks)";

After *ticks* seconds, *flag* will be set to TRUE.

suspendTimeouts ( ) causes all SIGALRM activity to be suspended. This routine should be used with care in those cases where critical code needs to be executed without interruption from **timeout ( )** or **alarm ( )** activity. All timeouts posted at the time the **suspendTimeouts ( )** call is performed, will be deferred until an **unsuspendTimeouts ( )** call is done. New **timeout ( )** activity can be generated after a suspend**Timeouts ( )** call is performed, thus allowing **sleep ( )** to work, but all **timeout ( )** activity prior to the **suspendTimeouts ( )** call is deferred.
suspendTimeouts ( ) can be called recursively up to the current maximum of five times. This is not recommended as the effect on deferred timeouts becomes hard to predict.

unsuspendTimeouts ( ) reinstates **timeout ( )** activity previously deferred by **suspendTimeouts (**). The timers are adjusted to account for the amount of time lost during the suspension, which might mean that one or more expired timers might go off as soon as **unsuspendTimeouts ( )** is executed.

Diagnostics

clkinit ( ) will return FAILURE if it is unable to allocate callout queues. **timeout ( )** returns FAILURE if the callout queues are full. **killtout ( )** returns FAILURE if it cannot find the specified function on the callout queues. **suspendTimeouts ( )** returns the number of suspended timeouts if the call succeeds and FAILURE if the maximum number of suspensions has been reached. **unsuspendTimeouts ( )** returns the number of reinstated timeouts if the call succeeds and FAILURE if there is not suspension to reinstate.

Caveats

It is recommended that you do not use the UNIX **alarm ( )** call if you are using **timeout ( )** or **sleep ( ),** since **timeout ( )** catches the SIGALRM signal and processes it. The clock routines attempt to avoid conflicts with other uses of **alarm ( )**, but should an alarm be set when **timeout ( )** or **sleep ( )** are invoked, processing of the alarm, should it expire during the **timeout ( )** or **sleep ( )** period will be deferred until the clock routines are finished using the SIGALRM signal.

All times given to the clock routines are in seconds and should be read as intervals of N–1 to N seconds in length. In other words, a time of 1 second is actually any amount of time from 0–1 seconds. It is strongly recommended that you never attempt a clock interval of less than 2 seconds for this reason.

The clock routines restore the original SIGALRM function whenever the callout table is empty and if there was an active **alarm ( )** when the clock routines took over SIGALRM, the **alarm ( )** is restarted minus the time that the clock routines used SIGALRM. If this means that the **alarm ( )** would have already expired, then processing is forced at this time.

sleep ( ) uses **longjmp ( )** to avoid problems when setting up short sleeps, where the **alarm ( )** might expire before the **pause ( )** is done. The use of **longjmp ( )** is not avoidable in the current implementation of UNIX, but because of it, sleep ( ) can delay other timeouts. In the worst cases, a **timeout ( )** might never be processed. The problem arises when two or more events are scheduled for the same clock tick. If one of the events is a **sleep ( )** and other events follow it, the events after the **sleep ( )** will not be executed until the next time SIGALRM is posted to the clock routines. If there are no events following the **sleep ( )** event with non-zero times, then no future SIGALRM will be posted and those events might never be processed. It is recommended that you do not use **sleep ( )** in conjunction with other clock activity if possible. If you avoid one second waits, you can use ti**meout ( )**, a flag, **setflag ( )**, and **pause ( )** to safely perform sleep activities. Any function called by the clock routines that performs a **longjmp ( )** runs the same risks that **sleep ( )** does.

See Also

list

ipc

Name

ipcInit, ipcOpen, ipcRecv, ipcSend, ipcClose, ipcRelease — Interprocess communication via named pipes

Synopsis

```
#include <sys/types.h>
#include <fcntl.h>
#include prismdefs.h
#include ipcComm.h

int ipcInit ( name,createMode )

char *name ;
int  createMode ;

int ipcOpen ( queue,name,accessMode )
struct IpcQ *queue ;
char *name ;
int accessMode ;

int ipcRecv ( queue )
struct IpcQ *queue ;

ipcSend ( queue,msg )
struct IpcQ *queue ;
char *msg ;
```

Description

These routines allow processes to communicate with each other using named pipes as the communication pathway. The messages passed between processes are expected to be ASCII lines terminated by a \n.

ipcInit allows a process to create a named pipe for communication purposes if it does not exist and if it does exist, test for another reader of the pipe. There should only be one reader of any pipe used by these routines. *name* is the pathname of the named pipe. *createMode* is the modes to be used when creating the named pipe, for example 0622, which would allow the owner to read the pipe and anyone else to write it. If *createMode* is 0, the pipe will not be created if it does not exist, but the check for a reader will be carried out if the pipe exists. The check for a read is accomplished by setting a *timeout* (see clock) and then attempting to open the pipe for writing. The open will not succeed if there is not a reader of the pipe.

ipcInit return FAILURE if the pipe exists and has a reader or it is unable to create the pipe. It returns SUCCESS if the pipe exists but has no reader or if it creates the pipe. SUCCESS means that it is legal to perform an ipcOpen. *errno* is always appropriately set either by UNIX or by ipcInit to indicate who returned FAILURE.

ipcOpen is a macro, which performs the opening of the named pipe *name* with access modes *accessMode* (see **fcntl (5)** & **open (2)**). The file description returned by the open is placed in the *queue* structure and returned as the return value.

ipcClose closes the named pipe associated with *queue*.

ipcRelease should be called if the *queue* structure for communication was a local variable. The buffer into which ipcRecv does its receives is allocated with malloc and needs to be returned if the *queue* structure will be destroyed by returning from the creating routine. It is recommended that *queue* structures be global or static to avoid the necessity of using ipcRelease.

ipcSend sends an ASCII *msg* through the named pipe associated with *queue*. *msg* should be terminated by a "\n\0" sequence. The message is transmitted with a 3-digit length appended to the beginning of the message. This implies that no message longer than 999 characters can be sent via these ipc mechanisms. **ipcRecv** strips this 3-digit length off the message before returning it. This 3-digit length allows transmission over the pipe to be atomic, with no chance of mixing of messages if there are multiple writers of the pipe. The message is always written as a single message, thereby guaranteed by UNIX to go onto the pipe as a single unit. It is read in two reads, first a 3 character read to get the length of the message that follows, and then a read of the **msg** itself.

ipcRecv reads the next message off its named pipe. It returns the length of the message as its return value. A return of 0 means the pipe is empty, which happens if the pipe was opened **NDELAY**, or if the pipe was in fact not a pipe, but instead a file or a tty and has reached **EOF**. A return of FAILURE indicates an interrupt or some problem.

The *queue* structure is as follows:

```
struct IpcQ
   {
      int    iq_queue ; /* System queue (file descriptor) */
      int    iq_flag ;
      int    iq_len ; /* Length of message */
      int    iq_size ; /* Size of the buffer */
   char *iq_buf ; /* Buffer in which to put message */
} ;
```

After performing an ipcRecv, *iq_buf* will point to the message and *ip_len* will be set to the length of the message.

Diagnostics

ipcInit returns FAILURE in the event of trouble or if the pipe is busy. In addition to standard UNIX errors when attempting to create the named pipe (see create (2)) it also sets errno.

ENOENT	The named pipe did not exist and **ipcInit** was not allowed to create it or was not able to create it.
EEXIST	Something other than a named pipe already existed with the name specified.
EBUSY	There already was a reader on the named pipe.

match

Name

match, exactMatch, matchNoCase, exactMatchNoCase — Searches a list of choices for the best match

Synopsis

```
#include  match.h


int match ( query,answers )
char *query ;
char **answers ;

int exactMatch ( query,answers )
char *query ;
char **answers ;

int matchNoCase ( query,answers )
char *query ;
char **answers ;

int exactMatchNoCase ( query,answers )
char *query ;
char **answers ;
```

Description

match ( ) searches down the list of possible *answers* looking for the best fit to *query*. It returns the index of the best match from 1 to N. If it cannot find a match, it returns *NO_MATCH*. If the query is ambiguous, it returns *AMBIGUOUS*. If the array of *answers* is defective, such as having the same entry more than once, it returns *BAD_DATABASE*. The *answers* array must contain a NULL pointer at the end to delimit the search.

exactMatch ( ) is the little brother of **match ( )**. It will return *NO_MATCH* unless there is an exact match in the list of *answers* with the *query* string. If there is an exact match in the list, it returns an index of from 1 to N indicating which answer in the list matched. **matchNoCase ( )** performs the same job as **match ( )** except that it ignores any upper/lower case information in determining whether things match or not. **exactMatchNoCase ( )** performs the same job as **exactMatch ( )** except that it ignores any upper/lower case information in determining whether things match or not.

options

Name

setDefaultOption, processOptionsFile, processOptions, setEnvVar, optStr, optInteger, optEnv, optBool, shellVariable, quotedShCmd, parseXeqY, setPrimaryOptionsFile, createStandardOptionsName, processProgramOptions, makeDfltOptionEnv, clearDfltOption — Routines for processing files containing "X=Y" pairs

Command Option

```
#include <stdio.h>
#include  prismdefs.h
#include  charBuffer.h
#include  arrays.h
#include  options.h

void setDefaultOption ( optPtr )
struct Option *optPtr ;

processOptionsFile ( file,optPtr )
char *file ;
struct Option *optPtr ;

processOptions ( linesPtr,optPtr )
char **linesPtr ;
struct Option *optPtr ;

void setEnvVar ( name,value )
char *name ;
char *value ;

void optStr ( op,name,value )
struct Option *op ;
char *name ;
char *value ;

void optInteger ( op,name,value )
struct Option *op ;
char *name ;
char *value ;

void optEnv ( op,name,value )
struct Option *op ;
char *name ;
char *value ;
```

```
void optBool ( op,name,value )
struct Option *op ;
char *name ;
char *value ;

char *shellVariable ( str )
char *str ;

char *quotedShCmd ( start,valuePtr )
char *start ;
char **valuePtr ;

int parseXeqY ( linePtr,xp,yp,remainderp )
char *linePtr ;
char **xp ;
char *yp ;
char *remainderp ;

void setPrimaryOptionsFile ( name )
char *name ;

char *createStandardOptionsName ( root,program )
char *root ;
char *program ;

void processProgramOptions ( program,optPtr )
char *program ;
struct Option *optPtr ;

void makeDfltOptionEnv ()

void clearDfltOption ()
```

Description

An option is something of the form *xxx=yyy*. The option is named *xxx* and has a value of *yyy*. The value portion can consist of ASCII characters, environment variables specified in normal shell syntax, quoted sections, and execution sections specified by the normal "`cmd`" shell syntax. For example, the following items are legal options:

```
UTMP=$PRISMROOT/etc/utmp
H=${HOME:-/}
DATE=‘date‘
```

How a specific option is used is specified by an option disposition structure of the form:

```
struct Option
  {
```

```
    char *o_name ;    /* Name of option */
    char *o_dflt ;    /* Default value of option */
  void (*o_hdlr)() ;  /* Function to handle option */
  POINTER o_adr ;     /* Address used by handler */
} ;
```

For default behaviors, o_name is NULL. In all other cases, the name must exactly match that of the option. **o_dflt** , if specified, will be substituted for the value of the option, if the option has no value. **o_hdlr** is the name of a function which will process the option. Four of the most common handlers are provided, **optStr**, **optInteger**, **optEnv**, and **optBool**.

optStr will place a copy of the value of an option in the address specified by *o_adr*. It does not attempt to free the previous value, since it may be a static rather than allocated item. **optInteger** will place the integer represented by the string value in the address specified by *o_adr*. **optEnv** will place the name of the option in the environment and give it the value of the option. *o_adr* is not required for the **optEnv** handler. **optBool** sets the integer specified by the address *o_adr* to either TRUE (1) or FALSE (0) based on *value*. If *value* is any of the following: T, TRUE, Y, YES, ON, or SET, the integer is set to TRUE. If *value* is any of the following: F, FALSE, N, NO, OFF, or CLEAR, the integer is set to FALSE. Also if *value* is a numerical string, the integer is set to TRUE if the number is not equal to zero. If it is equal to zero, then the integer is set to FALSE. The non-numerical values are case-insensitive, in other words they can be upper or lower case.

How *o_adr* is used depends on the handler. For the Dialogue handler used by the **clipFld (3x)** routines, *o_adr* points to an array of structures that the handler uses to process options.

setDefaultOption allows you to specify the option to be used when an option is encountered for which there is not a specific option disposition. The two most likely things to do are to either ignore the option entirely or place the option into the environment. These behaviors can be produced by calling **clearDfltOption**, which causes options with no specific disposition to be ignored, and **makeDfltOptionEnv**, which causes options without specific dispositions to be placed in the environment. If other behaviors are desired, an appropriate **Option** structure must be passed to **setDefaultOption**.

processOptionsFile processes the specified *file* with the options list specified by *optPtr* and whatever the currently active default behavior is.

processOptions is used interally to process a set of options that have already been read into an array of character pointer specified by *linesPtr*.

setEnvVar is used by **optEnv** to place values into the environment. It takes *name* and *value* and generates a string of the form *name=value* and places it in the environment.

shellVariable processes *str* as if it was a shell string and returns the converted string. The string returned must be freed when the caller is finished using it, since it is a **malloc() 'ed** string. The allowable syntax for string passed to *shellVariable* are ASCII characters, environment variables of the forms: **$XXX**, **${XXX}**, **${XXX:-yyy}**, **${XXX:=yyy}**, **${XXX:?yyy** , **${XXX:+yyy}** **@logID**, and shell command substrings of the form: 'cmd....' . The strings can also include sections within ""s or ”s and backslash quoting applies as well.

quotedShCmd handles shell commands within backquotes, that is, 'cmd...'. *start* must be pointing at the opening backquote character. **quotedShCmd** takes all the material inside the backquotes, passes them to a shell via *popen()*. *valuePtr* is the address of a character pointer, where a pointer will be placed that points to the resulting output from the shell. This pointer returned from **quotedShCmd** must be released when the caller is finished with it via a *free()* call. **quoteShCmd** returns a pointer to the closing backquote character in the input.

parseXeqY this routine takes as its input, lines of the form X=[Y], pointed to by **linePt**, and destructively parses them into the “X” portion to the left of the equal sign and the “Y” portion to the right of the equal sign. A pointer to the “X” portion is returned to the pointer pointed to by *xp*. A pointer to the “Y” portion is returned to the pointer pointed to by *yp*. Anything following the “Y” portion is pointed to by remainderp. TRUE is returned if the line has the form X=[Y]. FALSE is returned in other cases.

processProgramOptions processes one of two options files. If the user specified a primary options file, **via setPrimaryOptionsFile** and if that file exists, it is used. If it was not specified or does not exist, then **processProgramOptions** takes the name of the **program** and uses the base name portion (the portion to the right of the last '/' character, if any), and processes an option file with the name */etc/default/{basename}*. **createStandardOptionsName** generates a standard options file name based on a *root* directory and the *program* name. The generated name is of the form:

{root}/etc/default/{basename}

The returned value is an allocated string. Options which have a default value (an *o_dflt*) and which were not specified in the options file are set to their default values. This means that **processProgramOptions** guarantees that each value is either set to the default value or a value found in the options file.

Diagnostics

processOptionsFile and **processOptions** return FALSE if they are given no options array (or the options file does not exist or cannot be opened). They return TRUE in all causes where they are at least partially successful. *shellVariable* and *quotedShCmd* returns NULL if they are not given a string to translate otherwise they returns an allocated character buffer containing the translated string. The caller is responsible for freeing the translation. *parseXeqY* returns FALSE if it is not given a string to parse, if it is not given the addresses of two character pointers into which to place the beginning of the X and Y portions

of the answer, or if the string is not of the form *X=[Y]*, where *Y* can be optional. Serious consideration should be given to security issues if some of the options that are setable via *processProgramOptions* might allow alteration of the behavior of the program to subvert security measures programmed into an application program. For example, it is an attractive idea to have an application determine its root directory based on an environment variable so that the same program can run in more than one program environment simply by setting an environment variable. If the primary program options file was also to be found based on this root directory (see **createStandardOptionsName**) it would then be possible for a user to provide an alternate "root" directory structure with options of their own choosing. If the options relate to access issues, then this could be an invitation to tampering. For example, if debug options were made dynamic, so that they could be turned on and off via an options file, they might allow a user to get around the normal checks built into a program. The point is that some options should only be compile time options or guarantees should be built into the end application so that unprivileged people cannot subvert security.

Example

The following example processes an option file for a program with the default behavior being that entries are inserted in the environment. Notice how **LIMIT** is specified. This demonstrates how a program can be compiled with a particular default behavior, which is still dynamically controllable at run time. Notice that instead of compiling this code with.

-DLIMIT=100

which would be the normal way if the option was only a compile time option, it must be defined as a string instead:

-DLIMIT=\100

This allows the string to be placed in the Option structure and used to set the default value. This code will use at option file either from **/projectX/etc/default/{***program***}** or from **/etc/default/{***program***}**.

The code can be compiled with a different ROOTDBjust be specifying a different value: **DROOTDIR**=\"...\" on the compile line.

```
#include  prismdefs.h
#include  charBuffer.h
#include  arrays.h
#include  options.h

int debugLevel ;
char *mode ;
int limit ;
int indentFlag ;

#ifndef LIMIT
#define LIMIT "50"
#endif
```

```
#ifndef ROOTDIR
#define ROOTDB "/projectX"
#endif

struct Option progOpts[] =
  {
    {"DEBUG","0",optInteger,(POINTER)&debugLevel},
    {"MODE",(char*)NULL,optStr,(POINTER)&mode},
    {"LIMIT",LIMIT,optInteger,(POINTER)&limit},
    {"INDENT","Y",optBool,(POINTER)&indentFlag},
    {(char*)NULL,(char*)NULL,(void(*)())NULL,(POINTER)NULL}
  } ;
main(argc,argv)

int argc ;
char **argv ;

  {
    char *progName ;

    progName = *argv ;
    setPrimaryOptionsFile(createStandardOptionsName(ROOT
DIR,progName)) ;
    makeDfltOptionEnv() ;
    processProgramOptions(progName,&progOpts[0]) ;
    ...
  }
```

If the option file specified contained:

```
HISTORY=American
NATIONALITY=Japanese
MODE=edit
DEBUG=0x443
USER=$LOGNAME
```

the result would be that **mode** would point to a string containing “edit,” debugLevel would be set to the integer 0x443, and the environment would contain HISTORY=American, NATIONALITY=Japanese, and USER={logname}. limit would be set to 50, assuming that LIMIT was not defined when the code was compiled to some other value because LIMIT did not appear in the option file. *indentFlag* would be set to TRUE since INDENT did not appear in the option file and the default for INDENT was **Y**.

See Also

arrays
charBuffer
readLine
readFile

parseln

Name

parseln, parselnDA, localParselnDA, parselnEnhancedDA, localParselnEnhancedDA, sepln, & closeLocalParselnDA — Chop up strings into separate words

Synopsis

```
#include  parseln.h

int parseln ( ln,words,size )
char *ln ;
char **words ;
int size ;

int parselnDA ( ln )
char *ln ;

int localParselnDA ( dadp,ln )
int *dadp ;
char *ln ;

parselnEnhancedDA ( ln,sepChrs,quoteChrs )
char *ln ;
char *sepChrs ;
char *quoteChrs ;

localParselnEnhancedDA ( dadp,ln,sepChrs,quoteChrs )
int *dadp ;
char *ln ;
char *sepChrs ;
char *quoteChrs ;

sepln ( ln,words,size,quoteChr )
char *ln ;
char **words ;
int size ;
int quoteChr ;

int closeLocalParselnDA ( da )
int da ;
```

Description

parseln ( ) chops up the string pointed to by *ln*, separating each word with a NULL character. A word is either white space delimited or a string quoted with the ' ' or the '\" character. Within quoted words, the '\' character is the quote character, which can be used to quote ' ', '\", and '\' characters into the word. Pointers to the beginning of each word are placed in the *words* array. *size* specifies the size of the *words* array. **parseln ( )** returns the count of the number of words parsed from the *ln* string.

parselnDA ( ) performs the same task as **parseln ( )** except that it puts the resultant pointers to the individual words into a dynamic array, hence the caller never needs to worry about how big to make the words array. **parselnDA ( )** returns a dynamic array descriptor to an array of character pointers. If ln was NULL or pointed to the empty string, then it returns 0. Do not perform either **arrayClose ( )** or **arrayDone ( )** on the array descriptor returned by **parselnDA ( )**. Management of the dynamic array is handled entirely by parselnDA ( ). If it is desired to take over the array of character buffers associated with the returned array descriptor, use **arrayDetach ( )**. (See **arrays (3X)**)

parselnEnhancedDA ( ) performs the same task as **parselnDA ( )** except that the caller is allowed full control of the set of characters to be treated as separators between words and quoting characters. The separator list is specified by **sepChrs**. If it is not supplied, \t\n is used. The list of quote characters pairs is specified by **quoteChrs**. It must be an even number of characters in length otherwise it will be ignored. Each pair of characters is treated as a starting quote character and an ending quote character. For example, " [|]<|> " means that quoted sections appear within either "[|]"s or "<|>"s. If a quoted section begins and ends with the same character, then the **quoteChrs** should have this character appear twice, that is, ~ means the quoted sections start with a double quote and end with a double quote. This is the default behavior provided by **parseln ( )**. If **quoteChrs is NULL** or of zero length, then there will be no quoted sections permitted within the input. As with **parselnDA ( )**, **parselnEnhancedDA ( )** returns a dynamic array descriptor to the parsed words array. **parselnEnhancedDA ("ln, \t\n , \~\~ ")** is equivalent to **parselnDA (ln)**.

sepln ( ) performs the same function as **parseln ( )**, but specifies an alternate quoting character for quoted sections via **quoteChr. sepln (ln,words,size,' ')** has the same effect as **parseln (ln,words,size)**.

localParselnDA ( ) and **localParselnEnhancedDA ( )** perform the same operations as **parselnDA ( )** and **parselnEnhancedDA ( )** except that the calling routine must provide a pointer to an initialize array descriptor, *dadp*, which is used by **localParselnDA ( )** in which to store the parsed word. By initialized, it is meant that *dadp* points to an integer set initially to 0 before the first of a series of calls. It need not be modified after being initialized to 0 prior to the first call or after a call to **closeLocalParselnDA ( )**. If **localParselnDA ( )** is used, it is also the responsibility of the caller to call **closeLocalParselnDA ( )** with the array descriptor pointed to by *dadp* so that the array resources are released if the

array descriptor is truly local to a routine as opposed to a *static* or *global* variable. The extra work in using **localParseInDA ( )** is balanced by the fact that a library function can use **localParseInDA ( )** without risking the fact that some other routine is already using it. If two routines attempt to use either **parseInDA ( )** or **parseInEnhancedDA ( )** at the same time (meaning that both routines are expecting the information in the dynamic array returned by the **parseIn * DA ( )** routine to remain available while other routines are called), chaos ensues when the single dynamic array is released by later users. Since each user of **localParseIn * DA ( )** are using locally managed dynamic array descriptors, the information returned remains available and unaffected until either another call is performed or **closeLocalParseInDA ( )** is called.

Caveats

The *ln* string is modified by **parseIn ( )**, **sepIn ( )**, **parseInEnhancedDA ( )**, and **parseInDA**. Quoted strings are forced to terminate at the end of a line, therefore it is not possible to have a \n in a word.

Each successive call to **parseInDA ( )** or **parseInEnhancedDA ( )** overwrites the dynamic array of information. Be sure to copy the information or detach it, via , before calling either routine a second time if the data must be retained.

Be aware that the separators are “soft,” not “hard.” In other words, adjacent separators are treated as a single separator, not as a series of zero length words. For example:

“,,,,5,10,,,,15,15,,”

would return three words, 5, 10, and 15, if ’,’ was the separator.

readLine

Name

readLine, readContLine, readQuotedLine, resetReadLine (saveReadLineState), setReadLine (restoreReadLineState), setReadLineModes — Routines to read lines in from standard I/O streams that do not rely on preallocated buffers and which also keep count of the lines read

Synopsis

```
#include <stdio.h>
#include  prismdefs.h
#include  readInfo.h

extern int inlineCnt ;
extern int virtLineCnt ;

char *readLine ( fp )
FILE *fp ;

char *readContLine ( fp )
FILE *fp ;

char *readQuotedLine ( fp )
FILE *fp ;

void resetReadLine | saveReadLineState ( rip )
struct ReadInfo *rip ;

void setReadLine | restoreReadLineState ( rip )
struct ReadInfo *rip ;

void setReadLineModes ( modes )
int modes ;
```

Description

readLine ( ) reads a line from the stream specified by *fp*. It does the buffer management, unlike fgets ( ). For this reason the buffer is always large enough to hold whatever string is read. readLine ( ) returns a pointer to the line read.

readContLine ( ) is similar to readLine ( ) except that it automatically concatenates continuation lines together into one long virtual line. A line is considered to be followed by a continuation line when it ends in \<*nl*>. readContLine ( ) automatically removes the backslash and newline characters, unless the RI_LEAVE_BACKSLASH flag is set in the modes, and continues

reading the next line. Only upon encountering an unquoted newline character does readContLine ( ) return.

readQuotedLine ( ) is a further elaboration of readContLine ( ). It will concatenate lines ending in \<*nl*> , but will also read in quoted sections including the <nl> character as one virtual line. readQuotedLine ( ) only terminates a line when it finds the <*nl*> character outside of a quoted section and not preceded by a \ character. A quoted section can be surrounded by double quote, ", single quote, ', or accent grave, `.

readLine ( ), **readContLine ( )**, and **readQuotedLine ( )** update the global counters **inlineCnt** and **virtLineCnt**. **inlineCnt** is the count of the number of physical lines read. It is useful when reporting line numbers within files. It must be reset to 0 when you open a new file and start reading from it via **resetReadLine ( )** or **setReadLine ( )** and you must only use **readLine ( )**, **readContLine ( )**, or **readQuotedLine ( )** for the count to remain valid. *virtLineCnt* is the number of virtual lines returned by **readLine ( )**, **readContLine ( )**, or **readQuotedLine ( )**. For **readLine ( )**, *inlineCnt* and *virtLineCnt* are one and the same. For **readContLine ( )** and **readQuotedLine ( )**, *virtLineCnt* is incremented once each time it returns a concatenated line, whereas *inlineCnt* is incremented each time a character is encountered after a newline character.

resetReadLine ( ), also known as **saveReadLineState ( )**, resets the line counters for **readLine ( )**, **readContLine ( )**, and **readQuotedLine ( )**. It returns the previous values in the ReadInfo structure pointed to by *rip*, if *rip* is not **NULL**. It also saves the current working buffer being used by the **read * Line ( )** routines.

setReadLine ( ), also known as **restoreReadLineState ( )**, sets the line counters and the modes to those specified by the **ReadInfo** structure **rip** and restores the working buffer to what it was when the last **resetReadLine ( )** call was performed. **resetReadLine ( )** and **setReadLine ( )** might be used if more than one file was being accessed at the same time or if there is nesting of **read * Line ( )** usage, such as library routines. Any library routine that wants to read a file using any of the **read * Line ( )** routines should bracket its operation with **resetReadLine ( )** and **setReadLine ( )** calls so that any other routine using them at higher levels will not suddenly find the buffer it was working with wiped out.

setReadLineModes ( ) changes the modes being used by **readLine ( )** and **readContLine ( )** and returns the previous value. The default behavior of both routines is to retain the '\n' character at the end of each line. If the flag **RI_NO_NEWLINES** is set, the '\n' character is deleted from the end of each line as it is returned.

Diagnostics

readLine ( ), readContLine ( ), and readQuotedLine ( ) return NULL if the end of file is encountered

Caveats

readLine ( ), readContLine ( ), and readQuotedLine ( ) use a single buffer. You must copy the contents of the buffer if you need to do successive reads while retaining the previous information returned by earlier reads. If you are nesting file operations, use resetReadLine ( ) and setReadLine ( ) to save and restore the state information and the current working buffer.

inlineCnt and *virtLineCnt* will only be correct if all input from a stream is done with either readLine ( ), readContLine ( ), or readQuotedLine ( ) and if only one stream is read at a time and no seeks are performed. If input from more than one stream is required and *inlineCnt* and/or *virtLineCnt* are to be meaningful, it is the responsibility of the user to save and restore the various line counts for each stream before using the read routines again.

regEx

Name

regCmp, regPatSize, regularExp, regEx, startPatMatch, getSubString, getREname, regPrtPattern, regExprComputeLength, regExprCopy, regSubStringsCopy — Routines to compile regular expressions, execute them against ASCII strings, print the compiled patterns in C code style, and copy the associated structures

Synopsis

```
#include <stdio.h>
#include  prismdefs.h
#include  regularExp.h

char *regCmp ( re1[,...],(char *)NULL )
char *re1 ;

int regPatSize ( )

char *regularExp ( re,str,sspp )
char *re ;                  /* Compiled regular expression */
char *str ;                 /* ASCII input data */
struct SubStrings **sspp ; /* Where to return the substrings */

char *startPatMatch ( )     /* Return start of match location. */

char *getSubString ( index,ssp )
int index ; /* Substring of interest */
struct SubStrings *ssp    /* Description of substrings */

char *regEx ( re,str[,ss1,...] )
char *re ;          /* Compiled regular expression */
char *str ;         /* ASCII input data */
char *ss1 ;         /* Pointer to 1st substring buffer */

char *getREname ( chr,symbolicFlag )
int chr ;
int symbolicFlag ;

int regPrtPattern ( fp,arrayName,re,symbolicFlag )
FILE *fp ;
char *arrayName ; /* Name of C code array */
char *re ;          /* Compiled regular expression */
int symbolicFlag ; /* If TRUE, use symbolic names */

int regExprComputeLength ( rePtr )
```

char *rePtr ;

char *regExprCopy (re) ;
char *re ;

struct SubStrings *regSubStringsCopy (ssp)
struct SubStrings *ssp ;

Description

regCmp ( ) compiles a regular expression and returns a pointer to the compiled form of the expression. The compiled expression is placed in space allocated via the malloc ( ) routine. It is the responsibility of the caller of regCmp ( ) to free the regular expression when it is no longer needed. regCmp ( ) returns NULL if there is some problem with the expression such that it cannot be compiled.

Should the size of the compiled pattern be needed, *getPatSize ( )* returns the size of the compiled pattern as long as regCmp ( ) has not been called again before getPatSize ( ) is called. If the size of a regular expression is needed and regCmp () might have been called again or the pattern was precompiled, then use regExprComputeLength (), which scans the compiled regular expression and computes the length by determining where the end of the pattern is. If the pattern is defective, -1 is returned.

regularExp ( ) and regEx ( ) execute a compiled regular expression against ASCII input. regEx ( ) provides the original interface as described in the standard UNIX *Programmer's Reference Manual*. regularExp ( ) has a different interface making it safer to use and more appropriate where the patterns to be executed are not specified at compilation time. regEx ( ) arbitrarily returns substrings (see "(...)$n" below) detected by the compiled regular expression to the specified *ssN* buffer, where *N* is an index from 0–9. The size of each buffer is assumed to be large enough to hold however much must be returned. In a general case, where the pattern and perhaps the input, as well, are being specified at run time, knowing how big each buffer should be and how many substring buffers are required is impossible, so the program is always required to assume the worst case. regularExp ( ) provides a more graceful interface for these situations as well as one further enhancement to the language of substring extraction. regularExp ( ) returns substring information in a single allocated area (using malloc ( )) if the calling routine specifies the *sspp* argument. If the calling routine is not interested in the extracted substrings, then setting *sspp* to "(struct SubStrings **)NULL" causes the substring information to be discarded. As with the allocated space returned by regCmp ( ), the calling routine is responsible for freeing the allocated substring space with a free ("(char*)ssp"); once the substring information is no longer needed. regularExp ( ) also allows for support of "auto-indexed" substrings. This means that instead of specifying the index of the substring, the pattern assigns the index as each occurrence of "(...)$+" in the input pattern is satisfied. This allows patterns of the form:

"(...(...)$+...)+"

to assign as many substrings as are required for each repetition of the parent group. For example:

"([a-z]+([0-9]+0$+!)+"

applied against the string:

"This is an odd sequence: bob59!jennifer80!steve13!"

would extract 59, 80, and 13 and place them in substrings 10, 11, and 12.

The allocated space returned to the sspp pointer is to a variable length space that starts out with a structure whose size is determined by the number of substring pointers required by the data. It has the following layout:

Definition of structure	**Layout in memory**
struct SubStrings	
{	
int ss_cnt ;	**->count of substrings**
char *ss_substrings[1] ;	**->pointer to 1st substring**
} ;	**pointer to 2nd substring**
	. . .
*[1] is actually an [ss_cnt], since its size is determined by the number of substrings***.**	**pointer to n'th substring**
	1st substring (including '\0')
	2nd substring
	. . .
	n'th substring

getSubString ( ) is a macro that, when given an *index* and a pointer to the substring area, *ssp*, returned by regularExp ( ), returns a pointer either to the specified substring or to an empty string, " ". It always returns a valid pointer. If *ssp* is NULL or *index* is out of range or specifies a substring that has not been set, you get the empty string.

startPatMatch ( ) returns the location in the input string where the match started if the previous call to regEx ( ) or regularExp ( ) succeeded.

To assist in code generation of static patterns, regPrtPattern ( ) will print a compiled regular expression in one of two forms. If *symbolicFlag* is FALSE, the pattern is printed using hexidecimal values for opcode bytes of the pattern, meaning that the pattern that results does not require any headers to compile. If *symbolicFlag* is TRUE, the generated printing of the pattern uses the symbolic names described in regularExp.h and will require this header to compile. *fp* is the standard I/O stream on which the pattern is to be output, *arrayName* is the name of the C style array into which the printed pattern is to appear, and *re* is the compiled regular expression as returned by regCmp ( ). If *arrayName* is NULL, the pattern is printed without a name and the embracing "{...} ;" sequence.

getREname ( ) is used internally by regPrtPattern ( ), but is made available to programs that may want to use it to print pattern contents. It will interpret one opcode byte specified by *chr* into either a hexidecimal byte if *symbolicFlag* is FALSE or into a regular expression symbolic name if *symbolicFlag* is TRUE.

Making copies of compiled regular expressions requires knowing the length, which is not obvious from simple inspection. Compiled expressions are not normal C-strings,that is, a series of non-zero bytes terminated by a zero byte. Compiled expressions can contain zero bytes anywhere in the compiled expression. regExprCopy () makes allocated copies of compiled regular expressions. As with regCmp (), the caller is responsible for freeing the allocated space when appropriate. NULL is returned if the pattern is defective and hence cannot be copied.

regSubStringsCopy () makes allocated copies of the *SubStrings* structure returned by regularExp (). As with regularExp (), the caller is responsible for freeing the allocated space when appropriate.

The regular expression language is very similar to that used by ed, sed, and vi. There are a few differences. A description of the language follows:

- [...] — The list of characters within the braces specify the next valid input character. Each character may be listed explicitly or ranges may be specified, separated by the '-' character. The '-' character itself may only appear in the list at the beginning or the end so that it does not have its "range" meaning. Within range specification, the first character must come lexically before the second character, that is, [a–z], not [z–a]. Example: "[-.;a-z]" means that the '-', '.', ';', and all lower case letters are acceptable.
- [^...] — If the first character within the braces is '^', then any character EXCEPT those listed will match. "[^.;a-z-]" will match any character that is not a '.', ';', '-', or a lower case letter.
- ^ — If the first character of a regular expression is '^', it means the pattern must match from the beginning of the input (beginning of the line). Anywhere else in the expression, except as the first character of a character range (inside [|]s), '^' is a normal character.
- $ — If the last character of an expression is '$', it means the pattern must match to the '\n' character at the end of the line. Anywhere except the end of the pattern, '$' is a normal character.
- . — The '.' character matches any character expect '\n'.
- * — The '*' causes the immediately preceding character, character range, or group to match any number of times including 0. "a*[0–9]*" would match "aaaaaaaa", "aaaa12234343", "1342343232", or " ".
- + — The '+' character is the same as '*' except that the pattern it follows must match at least 1 time, hence "a+[a-9]+" would match "aaaa12234343", but none of the other examples in the previous description, since at least one 'a' and one digit is required.

- \ — This is the quoting character, causing the character following it to be treated as a normal character. This is necessary when attempting to search for characters with special meanings like '.'. \\x11 is required to search for the a \ itself. A '\' prior to a normal character, like 'a', is a no-op and is just discarded in the resulting pattern.
- {m} {m,} {m,n} — Integer values enclosed in "{^}" indicate the number of times the preceding character, character range, or group is to be repeated. m is the minimum number of matches required. *n* is the maximum number of matches required. If *n* is omitted, any number of matches greater than *m* is legal. If *m* appears without a ',' following it, then that number and only that number of matches is allowed, that is, "matches exactly m times." "(...)+" is equivalent to "(...){1,}" and "(...)*" is equivalent to "(...){0,}".
- (...) — Parentheses are used for grouping. The '*', '+', and "{m,n}" operations can be applied to patterns within parentheses, that is, "a+(ab+){3}" , which says one or more 'a' characters followed by three groups of 'a' followed by one or more 'b' characters - aaaaabbbabab.
- (...)$n (...)$+ — Parentheses followed by a '$' and either a digit, 09, or the '+' denote the substring extraction facility. Whatever portion of the input string that matches the description within the parentheses is copied to a substring. Using regEx ( ), only digits appearing after the '$' are meaningful and the substring is returned in the "n'th" ssN argument in the calling sequence. WARNING - the caller is totally responsible for ensuring that the "n'th" ssN buffer pointer has been supplied and is large enough to hold the extracted substring. Using regularExp ( ), both the digit form and the '+' form is meaningful. If a digit is supplied, then the "n'th" substring is set to the specified subpattern. If '+' is supplied, then regularExp ( ) automatically assigns the next substring to the extracted string starting at 10 and auto-indexing upwards. For example, if

  "(([a-z]+)$+boat[\t]+[a-z]+[\t]+)*"

  is applied to "the tugboat and sailboat on the river," then substring 10 would be "tug" and substring 11 would be "sail". A successful pattern match returns a pointer to the location AFTER the successfully matched information. If the pattern fails to match the input, "(char *)NULL" is returned.

Examples

The following example is a small program that finds each occurrence of a pattern specified by the user in a file and prints the lines containing the pattern with ">>" and "<<" enclosing each matched region. Substrings, if specified in the pattern, are ignored.

```
#include <stdio.h>
#include  regularExp.h
#include  mkcopy.h
#include  readInfo.h
main( )
```

```
  {
       char *pat,*re,*startp,*endp ;
       char *input ;
       char *file ;
       FILE *fp ;
       struct ReadInfo *ri ;
       int cnt,i ;

       for (;;)
        {
          printf( Pattern:  ) ;
          (void)setReadLineModes(RI_NO_NEWLINES) ;
          if ((pat = readLine(stdin)) == (char*)NULL)
           continue ;
          if ((re = regCmp(pat,(char*)NULL)) == (char*)NULL)
            {
             printf( %s — Pattern failed to compile.\n ,pat) ;
             continue ;
            }
          pat = mkcopy(pat) ;
          printf( File:  ) ;
          if (file = readLine(stdin))
            {
               file = mkcopy(file) ;
               if ((fp = fopen(file, r )) == (FILE*)NULL)
                  perror(file) ;
               else
                  {
                    resetReadLine(&ri) ;
                    (void)setReadLineModes(0) ;
                    for (cnt=0; input = readLine(fp) ;)
                      {
/*     Test for first occurrence of pattern in the line.*/

                       if (endp = regularExp(re,input,
                          (struct SubStrings **)NULL))
                          {
                          cnt++ ;
                          printf( Line: %d -  ,inlineCnt) ;
/*     Print out the line with >>{pat}<< around each matched sec-
tion. */

                          do
                            {
                             startp = startPatMatch( ) ;
                             i = startp - input ;
                             printf( %.*s ,i,input) ;
                             fputs( >> ,stdout) ;
                             i = endp - startp ;
                             printf( %.*s ,i,startp) ;
                             fputs( << ,stdout) ;
                             input = endp ;
                            }
                          while (endp = regularExp(re,input,
```

```
                                (struct SubStrings **)NULL)) ;
/*     Output the remainder of the line.*/
                               fputs(input,stdout) ;
                             }
                          }
                            printf( Lines matching pattern: %d\n ,cnt) ;
                            fclose(fp) ;
                            }
                         free(file) ;
                       }
                    free(re) ;/* Free compiled regular expression */
                    free(pat) ;
                  }
}
```

The following shorter example extracts the three parts of a phone number into substrings and prints them. Notice that the" "(^)" require '\'s preceding them and that it requires "\\" within a C program, since the C compiler also uses '\' as its quote character.

```
#include <stdio.h>
#include  regularExp.h
#include  mkcopy.h
#include  readInfo.h
main(argc,argv)
int argc ;
char **argv ;

    {
        char *re ;
        char *input ;
        FILE *fp ;
        struct SubStrings *ssp ;

        if (--argc != 1)
          {
          fprintf(stderr, Usage: findPhoneNos {file}\n ) ;
          return(1) ;
           }
        if ((fp = fopen(*++argv,r)) == (FILE *)NULL)
          {
           perror(*argv) ;
           return(2) ;
           }
        re = regCmp( \2-9][0-1][0-9])$0\\)
([0-9]{3})$1-([0-9]{4})$2 ,
           (char*)NULL) ;
        while (input = readLine(fp))
           {
           if (regularExp(re,input,&ssp))
           {
               printf( area code: %s exchange: %s line #: %s\n ,
                getSubString(0,ssp),
                getSubString(1,ssp),
              getSubString(2,ssp)) ;
          freecopy(ssp) ;/* Free substring storage */
         }
       }
     freecopy(re) ;
     return(0) ;
}
```

Caveats

- Free allocated space

  It is important that the calling program free the compiled patterns returned by regCmp ( ) and the substring area returned by regularExp ( ) when they are finished with them.

- Repeated substring extractions

  Repeating groups which contain "(...)$n" substring extractions will not work properly. For example, when **"(([a-z]+)$0BOAT)+"** is applied against **"tugBOATsailing"** will not have **"tug"** in substring 0, but will instead of "s" in $0. This results because the group was attempted a 2nd time, managed to find something that fit the class [a-z], but was unable to complete the pattern. (It took the whole word "sailing" and then backup character at a time hoping that **"BOAT"** was somehow buried in the material is had used to satisfy the "[a-z]+" portion of the pattern.) Unfortunately it also could not return to the previous successful state, since that had been wiped out. For cases like these, use the "auto-indexing" feature. Auto-indexed variable will not overwrite each other and the failed attempts are discarded so that "(([a-z]+)$+BOAT)+" is applied against "tugBOATsailing" would result in substring 10 being "tug" and that is all.

- Impossible patterns

  Some patterns cannot succeed even though they look as though they should. Two things must be kept in mind when writing patterns: 1) patterns always take as much input as they can, which is not what humans intuitively do and 2) patterns in which a part overlaps an immediately following part cannot be repeated more than once successfully. Such patterns will compile, but fail on some input. If the earlier part totally contains the following part as a subset, then the pattern will never succeed if more than one repetition is required regardless of the input. For example:

  "([a-z]+boat){2}"

  is such a pattern. You would think that applied against:

  "tugboatsailboat"

  that it should succeed since **"[a-z]+boat"** should match both **"tugboat"** and **"sailboat"** and the sum of them are two matches of the pattern. Unfortunately the word "boat" also matches the character class **"[a-z]"**. This means that the group matches **"tugboatsail"** as **"[a-z]+"** and only the final **"boat"** to the word **"boat"**, for one match. Then there is no more input and the pattern fails. The salient point to remember is that each subspecification matches as much as it can before it attempts more of the pattern.
  A .* , for example, will match the entire input. A pattern of **".*dog"** will work for a single match to the last word **dog** in the input, but any pattern requiring a minimum of two matches to the input will fail.

- Implementations differ

  This implementation of regCmp ( ) and regEx ( ), while including what the standard version does, are not the same. They have been corrected for some subtle problems with recursive patterns which include substring extractions and the extention for auto-indexing was added. Also malloc ( ) is used both for the compiler and the expression matcher.

See Also

ed(1)
lex(1)
malloc(3X)
regCmp(1X)

strmatch

Name

strmatch — Compares a target string with a pattern

Synopsis

```
#include  strmatch.h
int strmatch ( target,pattern )

char *target ;

char *pattern ;
```

Description

strmatch ( ) compares the *target* string to the *pattern* string and returns TRUE if the target matches the pattern and FALSE if it does not. The *pattern* is of the "sh" variety, not the "regular expression" variety. The *pattern* must COMPLETELY match the *target*. It is useful if you need to determine if some ASCII string matches some particular template. strmatch ( ) is NOT a partial matcher, where it looks for a pattern buried in a larger context and then tells you where that pattern is located in the larger context. For needs such as those, use the regular expression routines described in regcmp (3x) manual pages.

Meta-Characters

The following special sequences are supported by **strmatch ( )**:

*	Matches any sequence of 0 or more characters
?	Matches any single character
[xxx]	Matches any of the characters denoted by the string “xxx”
[!xxx]	Matches any character that is NOT one of the characters denoted by “xxx”
\n	The newline character
\r	The carriage return character
\b	The backspace character
\f	The formfeed character
\v	The vertical tab character
\t	The horizontal tab character
\NNN	The ascii character denoted by the octal value of the string “NNN”

Examples

Does "str" start with a digit and end in 'Z'?

```
"if (strmatch(str,"[0-9]*Z")) doSomething() ;"
```

Does "str" contain the name "John" somewhere in it?

```
“If (strmatch(str,”*John*”)) doSomething();”
“If (strmatch(strmatch(str,”*a???e*”)) do Something() ;”
```

Does "str" contain a five letter sequence starting with 'a' and ending with 'e'?

```
“If (strmatch(strmatch(str,”*a???e*”)) do Something() ;”
"if (strmatch(str,"*a???e*")) doSomething() ;"
```

See Also

match
regcmp

timeIncr

Name

timeIncr, fmtTimeIncr — Routine to convert an ASCII representation of a time increment into seconds and back

Synopsis

```
#include <sys/types.h>
#include <time.h>
#include  prismdefs.h
#include  timeDefs.h

long timeIncr ( str )
char *str ;

char *fmtTimeIncr ( incr,verbose )
unsigned long incr ;        /* Time increment in seconds */
int verbose ;               /* If TRUE, print in full English syntax */
```

Description

timeIncr ( ) converts an ASCII string comprised of the following type elements into a number of seconds: "NN weeks", "NN days", "NN hours", "NN minutes", and "NN seconds". The words may be abbreviated to any length, down to a single letter and the space between the number of the type identifier may be omitted, hence:

"2w 5d"

is an acceptable specification for 2 weeks 5 days. The order of objects is not relevent and any type of object can be omitted if not required to specify the time increment. No type of object can appear more than once in the specification and the entire object pointed to by *str* must correctly parse to be accepted. BADDATE (-1) is returned if parsing fails in any way.

fmtTimeIncr ( ) converts a number of seconds into its ASCII equivalent. If *verbose* is FALSE, the string generated will use the following abbreviations:

w	Weeks
d	Days
h	Hours
m	Minutes
s	Seconds

If *verbose* is not FALSE, then a full English form will be generated with commas and the word “and” included as is necessary. For example, if *incr* was 604801, the resulting string with *verbose* set to TRUE would be: “1 week and 1 second”. If *incr* was 90002, the resulting string would be “1 day, 1 hour, and 2 seconds”. Notice the plurals are also handled properly.

Examples

Spec	Value or Description
1s 1h	3601 seconds
2 weeks	14 days * 24 hours * 3600 seconds
5 min 30 sec	330 seconds
5m 3m	BADDATE — two minute specifications

Caveats

fmtTimeIncr ( ) creates its answer in a static buffer. A copy must be made of the answer before a subsequent call to fmtTimeIncr ( ) is made or the previous string will be lost.

Also note that the verbose form of output by fmtTimeIncr ( ) is not proper input to the timeIncr ( ) routine, while the terse form is. timeIncr ( ) does not understand the commas or the word and" used by the verbose form.

See Also

time(2)
ctime(3C)
tmtotime(3X)

tmtotime

Name

datetotime, jdaytotime, getime, asciiToTime, cvtTimeDefToJDay, getDSTdates, cvtJulian — Routines to convert tm structures and ASCII time specifications to UNIX time specifications

Synopsis

```
#include <sys/types.h>
#include <time.h>
#include  prismdefs.h
#include  timeDefs.h

time_t jdaytotime ( tmPtr )
struct tm *tmPtr ;

time_t datetotime ( tmPtr )
struct tm *tmPtr ;

int cvtJulianDay ( tmPtr )
struct tm *tmPtr ;

int getDSTdates ( year,jStartp,jEndp )
int year ;              /* Year of interest */
int jStartp ;           /* Where julian start day goes */
int jEndp ;             /* Where julian end day goes */

int cvtTimeDefToJDay ( year,tdp )
int year ;
struct TimeDef *tdp ;

time_t getime ( str,cntPtr,tmPtr )
char *str ;
int cntPtr ;            /* How many chars used. */
struct tm *tmPtr ;

time_t asciiToTime ( asciip,tmPtr )
char *asciip ;          /* ASCII description of date */
struct tm *tmPtr ;
```

Description

jdaytotime converts a partially filled structure of the the type *tm*, pointed to by *date*, into a long number of seconds equivalent to the system clock time (which is in GMT). It requires the element *tm_yday* be set. All other elements except the *tm_mon* and *tm_mday* can be optionally filled. If they are not set, they are set from the current date. It also completes filling in the day of the week, the month, day of the month, and setting the daylight savings time flag. The environment variable TZ is used to determine the time zone. If TZ is not available then Eastern Standard time is assumed.

jdaytotime returns a long which is equivalent to the system clock time OR in the cases of a bad conversion, BADDATE (-1). datetotime performs the same job as jdaytotime except that it requires the *tm_mon* and *tm_mday* to be set and not the *tm_yday*. The other elements may be optionally filled. It then completes the *tm* structure and returns a long with the equivalent UNIX system clock time or BADDATE.

cvtJulian generates only *tm_mon* and *tm_mday* from *tm_yday* in the *tm* structure pointed to by *tmPtr*.

getDSTdates determines the Julian day on which Daylight Savings Time begins and ends within the United States. The year of interest is specified by *year*. Account is taken of the Nixon years and the various changes in the law that has altered daylight savings time. *jStartp* points to the location where the starting Julian day is to be placed and *jEndp* points to the location where the ending Julian day is to be placed.

cvtTimeDefToJDay converts a *year* and a time of month specification, *tdp*, into a Julian day of the specified year. The time of month specification requires the index of the month (0–11), the week of the month (0 and up specify weeks from the start of the month, negative numbers specify from the end of the month), and the type of the week day of interest. BADDATE is returned if the specification cannot be satisfied otherwise the index of the Julian day is returned (0–365).

getime scans a character string, *str*, which points to a location which is supposed to contain an ASCII date. If it is a date in one of the following formats:

[weekday] mmm dd[,] yy[yy] hh:mm[:ss] tz
[weekday] mm/dd[/yy[yy]hh:mm[:ss] tz
[weekday] dd-mmm-yy[yy]
[weekday] mmddhhmm[yy] [tz]

or some variation of the first three formats, a system time is produced. If the date is somehow bad, **BADDATE** " (" -1) is returned. The order of the elements of the date can come in any order with the following restrictions: month must be followed by a day in the first two formats, or the day followed by a month in the third format, and the hours must be followed by minutes, but all the pieces of the date are optionally as long as some piece appears. For example:

10:30 1982 apr 2
saturday 1982 est april 10,82

are legal dates. The day of the week is superfluous, but if it is specified, the date will be in error if it does not match the day of the week specified by the remainer of the date. *tz* is a timezone. Currently *getime* understands **GMT**, **AST**, **ADT**, **EST**, **EDT**, **CST**, **CDT**, **MST**, **MDT**, **PST**, and **PDT**. The names of months, days of the week, and timezones can be any unique portion of the name, that is, Ap is sufficient to identify April. All upper and lower case differences are ignored. The original string is not affected by the parsing performed by getime.

If the date is successfully parsed, **cntPtr* is set to the number of characters parsed through the trailing white space after the date. If the date is embedded, adding the count to *str* will give you a new pointer to the first word after the date.

If *tmPtr* is not NULL, it will be used to return a fully parsed structure of the type *tm*.

asciiToTime () is an enhancement of getime (). It uses getime () to parse strings of the forms supported by getime (). If that is not successful, it attempts to convert dates of the form: “{locant} {weekday} in {month}”

A "{locant}" must be one of the following phrases:

1st or first

2nd or second

3rd or third

4th or fourth

5th or fifth

last or “next to last”

“{1st|2nd|3rd|4th|5th} {to|from} last”

“{first|second|third|fourth|fifth} {to|from} last”

All upper case characters are mapped to lower case and hence are not important in the syntax. If the specification is correct in this form, it is used to produce a UNIX time, which is returned as the answer from asciiToTime (), and a parsed *tm* structure is returned to the structure pointed to by *tmPtr*. If the ASCII representation cannot be parsed, **BADDATE** is returned. If a NULL pointer is provided for *tmPtr*, then the *tm* structure is not returned, but the routine operates properly and returns the UNIX time. The time returned will be relative to the current year if it is of the second form. The second form must have either four or six words in the phrase. Abbreviations are acceptable as long as they are unique. Periods(.) are ignored in the second form and so may be used as part of abbreviations.

The following date specifications are valid for asciiToTime ():

"2nd Tues. in Aug."

"Third Friday in March"

"next to last Mon in May"

Caveats

The integer pointed to by *cntPtr* will be advanced regardless of whether the date was good or bad. If the date was bad, this value should be ignored.

See Also

time()
ctime()

usage

Name

usage — Writes out a standard usage message plus a variable string as a form of help

Synopsis

```
#include  usage.h

extern char *usageMsg[] ;

void usage ( fmt,vargs... )
char *fmt ;
{type} vargs ;

setUsageExitValue ( val )
int val ;

char **setUsageMsg ( msg )
char **msg ;
```

Description

usage writes out the NULL terminated array of strings normally found in usageMsg when it is called and then exits with a non-zero exit value, normally 1. This provides a standard way to exit and provide help when a program is executed with improper arguments. If usage is supplied with a non-zero *fmt*, which is not the NULL string, it is passed to *vfmtStr* along with the arguments following to be printed out preceding the usage message. This allows the user to enhance the usage message with specific comments about the problem, if desired. If the messages printed are longer than 22 lines and stderr is connected to a tty type device, usage provides more paging-like behavior so that the user can easily read the message without having it scroll off the screen.

setUsageExitValue allows the user to specify an exit value after the usage message is printed that is different from the default value, 1. It returns the previous exit value.

setUsageMsg allows the user to specify an alternate usage message to be printed in place of the specified by usageMsg. usageMsg must be supplied by the user, but if there might be more than one usage message, then the alternate messages could be substituted in place of the usageMsg version using setUsageMsg.

Diagnostics

usage does not return. After printing the messages, it exits with either 1 or the value last specified by setUsageExitValue. setUsageExitValue returns the previous exit value. setUsageMsg returns a pointer to the previous usage message.

Caveats

usageMsg must be provided by the application code. The easiest way to produce such a message is via the texToFmt tool, which will convert clear ASCII text into an appropriate array of character strings suitable for compiling with the C compiler.

See Also

texToFmt(1x)
charBuffer(3x)

Abbreviations

A

AC
Alternating current

ACD
Automatic call distributor

AD
Application Dispatch

AD-API
Application dispatch application programming interface

ADPCM
Adaptive differential pulse code modulation

ADU
Asynchronous data unit

AGL
Application generation language

ALERT
VIS Alerter process

ANI
Automatic number identification

API
Application programming interface

ARU
Alarm relay unit

ASAI
Adjunct/Switch Application Interface

ASCII
American Standard Code for Information Interchange

ASI
Analog switch integration

B

BB
Bulletin board

bps
Bits per second

BRDG
Call bridging process

BSC
Binary synchronous communication

C

CCA
Call classification analysis

CDH
Call data handler

CELP
Continuously Excited Linear Prediction

CGEN
Voice system general message class

CICS
Customer Information Control System

CMP
Companion circuit card

CMS
Call Management System

CO
Central office

CPE
Customer provided equipment or customer premise equipment

CPN
Calling party number

CPT
Call progress tones

CPU
Central processing unit

CSU
Channel service unit

CVS
Converse vector step

D

dB
Decibels

DB
Database

DBC
Database checking process

DBMS
Database management system

DC
Direct current

DCE
Data communications equipment

DCP
Digital communications protocol

DIO
Disk input and output process

DIP
Data interface process

DMA
Direct memory access

DNIS
Dialed number identification service

DSP
Digital signal processor

DTE
Data terminal equipment

DTMF
Dual tone multi-frequency

DTR
Data terminal ready

E

EBCDIC
Extended Binary Coded Decimal Interexchange Code

EIA
Electronic Industries Association

EISA
Extended Industry Standard Architecture

EMI
Electromagnetic interference

ESD
Electrostatic discharge

ESDI
Extended Serial Data Interface

ESS
Electronic Switching System

ET
Error tracker

EXTA
External alarms feature message class

F

FCC
Federal Communications Commission

FDD
Floppy disk drive

FEP
Front end processor

FFE
Form Filler Plus feature message class

FIFO
First-in-first-out processing order

foos
Facility out-of-service state

FTS
File transfer process message class

G

GEN
PRISM logger and alerter general message class

GSE
Graphical Speech Editor

GUI
Graphical user interface

H

HDD
Hard disk drive

HLLAPI
High Level Language Application Programming Interface

HOST
Host interface process message class

hwoos
Hardware out-of-service state

Hz
Hertz

I

IBM
International Business Machines

ICK
Integrity checking process message class

ID
Identification

IDE
Integrated Disk Electronics

IE
Information element

INIT
Voice system initialization message class

inserv
In-service state

IPC
Interprocess communication

IPC
Intelligent Ports Card (IPC-900)

IPCI
Integrated personal computer interface

IRAPI
Intuity Response Application Programming Interface

IRQ
Interrupt request

ISA
Industry Standard Architecture

ISDN
Integrated Services Digital Network

ISV
Independent Software Vendor

ITAC
International Technical Assistance Center

IVP4
Integrated Voice Processing card with 4 analog channels

IVP6
Integrated Voice Processing card with 6 analog channels

IVPSS
Integrated Voice Processing System Software

K

Kbps
Kilobites per second

Kbyte
Kilobyte

L

LAN
Local area network

LDB
Local database

LED
Light-emitting diode

LIFO
Last-in-first-out processing order

LN
Load number

LOG
VIS logger process message class

LST1
Line side T1

LU
Logical unit

M

manoos
Manually out-of-service state

MAP/100
Multi-Application Platform 100

MAP/100C
Multi-Application Platform 100C

MAP/40
Multi-Application Platform 40

Mbps
Megabits per second

Mbyte
Megabyte

ms
Millisecond

msec
Millisecond

MHz
Megahertz

MTC
Maintenance process

N

NCP
Network Control Program

NEBS
Network Equipment Building Standards

NEMA
National Electrical Manufacturers Association

netoos
Network out-of-service state

NFAS
Non-Facility Associated Signaling

NFS
Network file sharing

NMVT
Network Management Vector Transport

NM-API
Network Management - Application Programming Interface

nonex
Nonexistent state

NRZ
Non Return to Zero

NRZI
Non Return to Zero Inverted

O

OEM
Original equipment manufacturer

OGA
Operator generated alert

P

PBX
Private branch exchange

PC
Personal computer

PCB
Printed circuit board

PCM
Pulse code modulation

PEC
Price element code

PRI
Primary rate interface

PSTN
Public switch telephone network

PS&BM
Power supply and battery module

R

RAM
Random access memory

RECOG
Speech recognition feature message class

RDBMS
ORACLE relational database management system

REN
Ringer equivalence number

RFS
Remote file sharing

RM
Resource manager

RMB
Remote maintenance board

RTS
Request to send

S

SBC
Sub-band coding

SCCS
Switching Control Center System

SCSI
Small Computer System Interface

SDLC
Synchronous Data Link Control

SDN
Software Defined Network

SID
Station identification

SIMM
Single inline memory module

SLIP
Serial Line Interface Protocol

SNA
Systems Network Architecture

SNMP
Simple Network Management Protocol

SP
Signal processor circuit card

SPIP
Signal processor interface process

SPPLIB
Speech processing library

SQL
Structured Query Language

SR
Speech recognition

SYS
UNIX system calls message class

sysgen
System generation

T

tas
Transaction assembler

TCC
Technology Control Center

TCP/IP
Transmission control protocol/internet protocol

TDM
Time division multiplexing

TE
Terminal emulator

THR
Threshold message class

TKR
Token Ring

TLI
Transport layer interface

TLP
Transmission level plan

T/R
Tip/Ring circuit card

TRIP
Tip/Ring interface process

TSO
Technical Service Organization

TSO
Time Share Operation

TSM
Transaction state machine process

TTS
Text-to-Speech

TWIP
T1 interface process

U

UK
United Kingdom

USOC
Universal service ordering code

UVL
Unified Voice Library

V

VDC
Video display controller

VIS
Intuity CONVERSANT Voice Information System

VPC
Voice processing comarketer

VRU
Voice response unit

VROP
Voice response output process

Glossary

Numerics

3270 interface
A link between one or more Intuity CONVERSANT Voice Information System (VIS) machines and a host mainframe. In Intuity CONVERSANT VIS documentation, the 3270 interface means the link between one or more VIS machines and an IBM host mainframe.

4ESS
A large AT&T central office switch used to route calls through AT&T's telephone network.

A

ACD
See "automatic call distributor."

ADPCM
See "adaptive differential pulse code modulation."

adaptive differential pulse code modulation
A means of encoding analog voice signals into digital signals by adaptively predicting future encoded voice signals. This adaptive modulation method reduces the number of bits required to encode voice. See also "pulse code modulation."

adjunct products
Products (for example, Adjunct/Switch Application Interface) that the Intuity VIS administers via cut-through access to the inherent management capabilities of the product itself; this is in opposition to CONVERSANT VIS's ability to administer the switch directly.

Adjunct/Switch Application Interface
An optional feature package that provides an Integrated Services Digital Network-based interface between AT&T PBX's and adjunct processors.

affiliate
A business organization that AT&T controls or which with AT&T is in partnership.

alarm relay unit
A unit used in central office telecommunication arrangements that transmits warning indicators from telephone communications equipment (like the Intuity CONVERSANT VIS) to audio.

alerter
A system process that responds to patterns of events logged by the "logdaemon" process.

analog
An analog signal, such as voice or music, that varies in a continuous manner. An analog signal may be contrasted with a digital signal, which represents only discrete states.

application
Made of several components that provide an automated version of the communication between a caller and an attendant. The Intuity CONVERSANT VIS provides several methods for creating applications, including Script Builder, the Intuity Response Application Programming Interface (IRAPI), and transaction state machine (TSM) script language.

application administration
The component of the Intuity CONVERSANT VIS that provides access to the applications currently available on your system and helps you to manage and administer them.

application installation
A two-step process in which the Intuity CONVERSANT VIS invokes the TSM script assembler for the specific application name and files are moved to the appropriate directories.

application verification
A process in which the Intuity CONVERSANT VIS verifies that all the components needed by an application are complete.

ASCII
An acronym for American Standard Code for Information Interchange, a standard for data representation. ASCII code represents alphanumeric characters as binary numbers. The code includes 128 upper- and lowercase letters, numerals, and special characters. Each alphanumeric and special character has an ASCII code (binary) equivalent that is 1 byte long.

asynchronous communication
A method of data transmission in which bits or characters are sent at irregular intervals and are spaced by start and stop bits and not by time. See also “synchronous communication.”

asynchronous data unit
An electronic communications device that allows computer systems to communicate over asynchronous lines more than 50 feet in length.

AUDIX Voice Power
A complete voice-mail messaging system accessed and operated by touch-tone telephones and integrated with a switch or “Private Branch Exchange.”

automatic call distributor
A telephone system that recognizes and answers incoming calls and completes these calls based on a set of instructions contained in a database. The Automatic Call Distributor can send the call to an operator or group of operators as soon as the operator has completed a previous call or after the system has played a message to the caller.

automatic number identification
A method of identifying the calling party by automatically receiving a string of digits that identifies the calling station of a particular customer.

B

back up
The preservation of the information in a file in a different location, so that the data is not lost in the event of hardware or system failure.

backing up an application
A utility that makes an archive copy of a completed application or makes an interim copy of an application in progress. The backup copy can be restored to the VIS if the online version is damaged, or if you make revisions and wish to go back to the previous version.

barge-in
A capability provided by WholeWord speech recognition that allow callers to speak their responses to the VIS prompt and have those responses recognized before the prompt has finished playing.

batch file
A file containing one or more lines, each of which is a command executable by the UNIX shell.

binary synchronous communications
A character-oriented synchronous link protocol.

blind transfer protocol
A protocol in which a call is completed as soon as the extension is dialed, without having to wait to see if the telephone is busy or if the caller answered.

bridging
The process of connecting one telephone network connection to another telephone network connection over the Intuity CONVERSANT VIS TDM bus. Bridging decreases the processing load on the system since an active bridge does not require speech processing, database access, host activity, etc., for the transaction.

BSC
See “binary synchronous communication.”

bundle
In the context of the Enhanced File Transfer package, this term is used to denote a single file, a group of files (package), or a combination of both.

byte
A unit of storage in the computer. On many systems, a byte is 8 bits (binary digits), the equivalent of one character of text.

C

call classification analysis
An optional feature package that allows application developers to classify the disposition of originated and transferred calls.

call data event
A parameter that specifies a list of variables that are appended to a call data record at the end of each call.

call data handler process
A software process that accumulates generic call statistics and application events.

called party number
The number dialed by someone making a telephone call. It can be used by telephone switching equipment to selectively route an incoming call to a particular department or agent.

caller
The party that calls for a service, gets connected to the Intuity CONVERSANT VIS, and interacts with the system. As the Intuity CONVERSANT VIS is also capable of making outbound calls for service, the caller can also be the person who responds to those outbound calls.

call progress tones
Standard telephony sounds that indicate the status of the call. These sounds include busy, fast busy, ringback, reorder, etc.

card cage
An area within a Intuity CONVERSANT VIS platform that contains and secures all of the standard and optional circuit cards used in the system.

cartridge tape drive
A high-capacity data storage/retrieval device that can be used to transfer large amounts of information onto high-density magnetic cartridge tape based on a predetermined format. This tape can be removed from the system and stored as a backup, or used on another system.

caution
An admonishment used when there is a possibility of a service interruption or a loss of data.

CCA
See “call classification analysis.”

CDH
See “call data handler process.”

central office
An office or location in which large telecommunication machines such as telephone switches and network access facilities are maintained. These locations follow strict installation and operation requirements.

central processing unit
A component of the Intuity CONVERSANT VIS that is based on either the Multi-Application Platform 100 (MAP/100), MAP/40, or MAP/100C.

channel
See “port.”

CICS
See “Customer Information Control System.”

circuit card upgrade
A new circuit card that replaces an existing one in the platform. Usually the replacement is an updated version of the other card, and the replacement is designed to deal with technology made obsolete by industry trends or a new VIS release.

cluster controller
A bisynchronous interface that provides a means of handling remote communication processing.

command
An instruction or request given by the user to the VIS software to perform a particular function. An entire command consists of the command name and options.

CompuLert/SCCS interface
An optional feature that enables remote or console monitoring of error messages generated from the Intuity CONVERSANT VIS. CompuLert is a centralized maintenance system for monitoring minicomputers, computer mainframes, etc. The Switching Control Center System (SCCS) is similar to the CompuLert system, but is used to support 4ESS local switching systems.

configuration
The arrangement of the software and hardware of a computer system or network. The Intuity CONVERSANT VIS configuration includes either a standard or custom processor, peripheral equipment (for example, printers, modems), and software applications. Configuration also refers to the way the switch network is set up; that is, the types of products that are in the network and how those products communicate.

configuration management
The component of the VIS that allows you to manage the current configuration of voice channels, host sessions, and database connections, assign scripts to run on specific voice channels or host sessions assign functionality to SP and T1 cards, and perform various maintenance functions.

Converse Data Return (conv_data)
A Script Builder action that supports the DEFINITY call vectoring (routing) feature by enabling the switch to retain control of vector processing in the VIS environment. It supports the DEFINITY "converse" vector command to establish a two-way routing mechanism between the switch and the VIS to facilitate data passing and return.

controller circuit card
A circuit card used on a computer system that controls its basic functionality and makes the system operational. These cards are used to control magnetic peripherals, video monitors, and basic system communications.

copying an application
A utility in which information from a source application is directed into the destination application.

coresidency
The ability of two products or services to operate and interact with each other on a single hardware platform. An example of this is the use of AUDIX Voice Power along with Intuity CONVERSANT on the same VIS platform.

CPU
See "central processing unit."

crash
An interactive utility for examining the operating system core and for determining if system parameters are being exceeded.

custom speech
Unique words or phrases to be used in Intuity CONVERSANT VIS voice prompts that AT&T records for a customer on a custom basis.

custom vocabulary
A specialized package of unique words or phrased created on a per-customer basis and used by WholeWord or FlexWord speech recognition.

Customer Information Control System
Part of the operating system that manages resources for running applications (for example, IND$FILE). Note that TSO and CMS provide analogous functionality in other host environments.

D

danger
An admonishment used when there is a possibility of personal injury.

data interface process
A software process that communicates with Script Builder applications.

database
A structured set of files, records, or tables.

database field
A field used to extract values from a local database and form the structure upon which a database is built.

database table
A structure, made up of columns and rows, that holds information in a database. Database tables provide a means of storing information that changes too often to "hard-code," or permanently store, in the transaction outline.

debug
The process of locating and correcting errors in computer programs. This process is also referred to as "troubleshooting."

default
The way a computer performs a task in the absence of other instructions.

default owner
The owner of a channel when no process takes ownership of that channel. The default owner holds all idle, in-service channels. In terms of the IRAPI, this is typically the Application Dispatch process.

diagnose
The process of performing diagnostics on Tip/Ring, T1, or SP circuit cards or a bus.

dialed number identification service
A service that allows incoming calls to contain information about the telephone number for which it is destined.

directory
A type of file used to group and organize other files or directories.

DNIS
See "dialed number identification service."

DIP
See "data interface process."

display errdata
A command that displays system errors sent to the logger.

DTMF
See “dual tone multi-frequency.”

dual 3270 links
A feature that provides an additional physical unit (PU) to allow a cost-effective means of connecting to two host computers. The customer can connect a VIS to two separate FEPs or to a single FEP shared by one or more host computers. Each link supports a maximum of 32 LUs.

dual tone multi-frequency
A touch tone.

dump space
An area of the disk that is fixed in size and should equal the amount of RAM on the system. The operating system “dumps” an image of core memory upon system crashes. The dump can be fetched after rebooting for analysis of what may have caused the crash.

E

editor system
A system that allows speech phrases to be displayed and edited by a user. See “Graphical Speech Editor.”

Enhanced File Transfer
A feature that allows the transferring of files automatically between the Intuity CONVERSANT VIS and a synchronous host processor on a designated logical unit.

Enhanced Serial Data Interface
A software- and hardware-controlled method used to store data on magnetic peripherals.

error message
A message on the screen indicating that something is wrong and possibly suggesting how to correct it.

Error Tracker process
See “etStub.”

Ethernet
A name for a local area network that uses 10BASE5 or 10BASE2 coaxial cable and InterLAN signaling techniques.

etStub
A system process that processes pre-Version 3.1 error message logging requests. These requests are transformed and passed on to the “logdaemon” process.

event
The notification given to an application when some condition occurs.

external actions
Specific tasks and interfaces controlled by Intuity CONVERSANT VIS software that allow a Script Builder application script to invoke processes and interact with other products or services. For example, a Intuity CONVERSANT VIS application script can invoke AUDIX Voice Power functionality through the used of an external action within an application script.

F

feature
A function or capability of a product or an application within the Intuity CONVERSANT VIS.

feature package
An optionally purchased package that may contain both hardware and software resources, which provides additional functionality to a standard system.

feature_tst script package
A standard CONVERSANT VIS software program that allows a VIS user to perform self-tests of critical hardware and software functionality.

field
A “slot” in a VIS window that holds one column of information in a row.

file
A collection of data treated as a basic unit of storage.

file transfer
An option that allows you to transfer files interactively or directly to and from UNIX using the File Transfer System.

filename
Alphabetic characters used to identify a particular file.

FlexWord speech recognition
A type of speech recognition based on subword technology that recognizes phonemes or parts of words of American English vocabularies. See “subword technology.”

Form Filler Plus
An optional feature package that provides the capability for application scripts to record caller’s responses to prompts for later transcription and review.

function key
A key, labeled F1 through F8, on your keyboard to which the Intuity CONVERSANT VIS software gives special properties for manipulating the user interface.

G

Graphical Speech Editor
A window-driven, X Windows/Motif based, graphical user interface (GUI) that can be accessed to perform different functions associated with the creation and editing of speech files to be used by VIS applications.

H

hard disk drive
A high-capacity data storage/retrieval device that is located inside a computer platform. A hard disk drive stores data on nonremovable high-density magnetic media based on a predetermined format for retrieval by the system at a later date.

hardware
The physical components of a computer system. The central processing unit, disks, tape and floppy drives, etc., are all hardware.

hardware upgrade
Replacement of one or more fundamental platform hardware components (for example, the CPU or hard disk drive), but the existing platform and other existing optional circuit cards remain.

High Level Language Applications Programming Interface (HLLAPI)
An application programming interface that allows user to write custom applications that can communicate with the host via an API.

HLLAPI
See “High Level Language Applications Programming Interface.”

host computer
A computer linked to a network providing a range of services, such as database access and computation. The host computer operates in a time-sharing manner with other computers linked to it via the network.

I

iCk
The system integrity checking process.

idle channel
A channel that either has no owner or is owned by its default owner and is onhook.

IND$FILE
The standard SNA file transfer utility that runs as an application under CICS, TSO, and CMS. IND$FILE is independent of link-level protocols such as BISYNC and SDLC.

indexed table
A table that, unlike a nonindexed table, can be searched via a field name that has been indexed.

initialize
To start up the system for the first time.

Integrated Services Digital Network
A network that provides end-to-end digital connectivity to support a wide range of voice and data services.

Integrated Voice Processing circuit card
The IVP4 or IVP6 circuit card.

intelligent transfer protocol
A transfer protocol that monitors the line after dialing is complete to determine whether a busy, reorder (fast busy), or other failure has been encountered. It also recognizes when the extension is answered or if the extension is not answered after a specified number of rings.

interface
The access point of a system. With respect to the Intuity CONVERSANT VIS, the interface is designed to provide you with easy access to the software's capabilities.

interrupt
The termination of voice and/or telephony functions when some condition occurs.

Intuity Response Application Programming Interface
A library interface the provides a standard development interface for voice-telephony applications.

ipcs
A command that reports interprocess communication facilities status.

IRAPI
See "Intuity Response Application Programming Interface."

ISDN
See "Integrated Services Digital Network."

K

keyboard mapping
In emulation mode, this feature enables the keyboard to send 3270 keyboard codes to the host according to a configuration table set up during installation.

keyword spotting
A capability provided by WholeWord Speech Recognition that allows the VIS to recognize a single word in the middle of an entire phrase spoken by a caller in response to a prompt.

L

LAN
See "local area network."

library states
The state information about channel activities maintained by the IRAPI.

line side T1
A digital method of interfacing a Intuity CONVERSANT VIS to a PBX or switch using T1-related hardware and software.

listfile
An ASCII catalog that lists the contents of one or more talkfiles. Each application script is typically associated with a separate listfile. The listfile maps speech phrase strings used by application scripts into speech phrase numbers.

local area network
A data communications network in a limited geographical area. The local area network provides communications between computers and peripherals.

local database
A database residing on the Intuity CONVERSANT VIS.

logical unit
A type of SNA Network Addressable Unit.

logdaemon
System information and error logging process.

logger
See “logdaemon.”

logging on/off
Entering or exiting the Intuity CONVERSANT VIS software.

LU
See “logical unit.”

M

magnetic peripherals
Data storage devices that use magnetic media to store information. Such devices include hard disk drives, floppy disk drives, and cartridge tape drives.

main screen
The Intuity CONVERSANT VIS VERSION 5.0 screen from which you are able to enter System Administration or Voice System Administration.

maintenance process
A software process that runs temporary diagnostics.

Manual Configurator Program
A software program that resolves or blocks the allocation of CPU and memory resources for controlling and optional circuit cards.

masked event
An event that an application can ignore (that is, the application can ask not to be informed of the event).

master
A board that provides clock information to the TDM bus.

megabyte
A unit of memory equal to 1,048,576 bytes (1024 x 1024). It is often rounded to one million.

Microsoft
A company that manufactures software products, primarily for IBM-compatible computers.

mirroring
A method of data backup that allows all of the data transactions to the primary hard disk drive to be copied and maintained on a second identical drive in near real time. If the primary disk drive crashes or becomes disabled, all of the data stored on it (up to 1.2 billion bytes of information) is accessible on the second mirrored disk drive.

MS-DOS
A personal computer disk operating system developed by the Microsoft Corporation.

MTC
See “maintenance process.”

multi-threaded application
A single process/application that controls several channels. Each thread of the application is managed explicitly. Typically this means state information for each thread is maintained and the state of the application on each channel is tracked.

N

NetView
An optional feature package that transmits high-priority (major or critical) messages to the host as Operator-Generated Alerts (OGAs) over the 3270 host link. The NetView Alarm feature package does not require a dedicated LU.

new error logging environment
A more flexible and informative environment for logging errors and status messages (introduced in CONVERSANT VIS Version 3.1). Customer applications created earlier than V3.1 that log messages require conversion to this new environment.

new operating system
The UnixWare operating system being introduced in Intuity CONVERSANT VIS V5.0.

nonindexed table
A table that may be searched only in a sequential manner and that cannot be searched via a field name.

nonmasked event
An event that must be sent to the application. Generally, an event is nonmaskable if the applicaiton would likely encounted state transition errors by trying to ignore the event.

null value
An entry containing no value. A field containing a null value is normally displayed as blank and is different from a field containing a value of zero.

O

obsolete hardware
Hardware that is no longer supported on Intuity CONVERSANT VIS V5.0.

on-line help
Messages or information that appear on the user's screen when a "function key" (F1 through F8) is pressed.

Operator Generated Alerts
System monitoring messages transmitted from the CONVERSANT VIS or other computer system to an IBM host computer that are classified as critical or major.

option
An argument used in a command line to modify program output by modifying the execution of a command. When you do not specify any options, the command will execute according to its default options.

ORACLE
A company that produces Relational Database Management software. It is also used as a generic term that identifies a database residing on a local or remote system that is created and maintained using an ORACLE RDBMS product.

P

PBX
See "private branch exchange."

PCM
See "pulse code modulation."

peripheral (device)
Equipment such as printers or terminals that is in addition to the basic processor.

permanent process
A process that starts and initializes itself before it is needed by a caller.

phoneme
A single basic sound of particular spoken language. The English language contains 40 phonemes that represent all basic sounds used with the language. As an example, the word "one" can be represented with three phonemes, "w" - "uh" - "n." Phonemes vary between languages because of guttural and nasal inflections and syllable constructs.

phrase filtering
The rejection of unrecognized speech. The WholeWord and FlexWord speech recognition packages can be programmed to reprompt the caller if the spoken response was not recognized by the VIS.

phrase tag
A string of up to 50 characters that identify the contents of a speech phrase used by an application script.

platform migration
See “platform upgrade.”

platform upgrade
The process of replacing the existing platform with a new platform.

poll
A message sent from a central controller to an individual station on a multipoint network inviting that station to send if it has any traffic to send.

polling
A network arrangement whereby a central computer asks each remote location whether they wish to send information. This arrangement enables each user or remote data terminal to transmit and receive information on shared facilities.

port
A connection or link between two devices that allows information to travel to a desired location. See “telephone network connection.”

Primary Rate Interface
An optional feature package that provides a digital interface capable both of receiving and originating telephone calls directly from/to an AT&T 4ESS switch.

private branch exchange
A private switching system, either manual or automatic, usually serving an organization, such as a business or government agency, and usually located on the customer’s premises.

processor
In Intuity CONVERSANT VIS documentation, the computer on which UnixWare and Intuity CONVERSANT VIS software runs. In general, the part of the computer system that processes the data. Also known as the “central processing unit.”

ps
A command that shows active processes. This command displays the process table and can be used to determine which processes are consuming large amounts of system resources, such as CPU time.

pseudo driver
A driver that does not control any hardware.

pulse code modulation
A digital modulation method of encoding voice signals into digital signals. See also “adaptive differential pulse code modulation.”

R

recovery
The process of using copies of the VIS software to reconstruct files that have been lost or damaged. See also “restore.”

remote database
The component of the VIS that provides access to information not currently on the VIS.

remote maintenance board
A Intuity CONVERSANT VIS board that is equipped standard on all new MAP/100 and MAP/40 platform purchases. This card, available with a built-in modem, allows remote personnel (for example, field support) to access all Intuity CONVERSANT VIS machines with a standard simplified process.

reports administration
The component of the VIS that provides access to system reports, including VIS call classification reports, call data detail reports, call data summary reports, message log reports, and traffic reports. In addition, if AUDIX Voice Power R2.1.1 is installed on your system, the reports administration component gives you access to AUDIX Voice Power reports.

restore
The process of recovering lost or damaged files by retrieving them from available backup tapes or from another disk device. See also “recovery.”

restore application
A utility that replaces a damaged application or restores an older version of an application.

reuse
The concept of reusing an existing system component after a software upgrade or platform migration.

roll back
To cancel changes to a database since the point at which changes were last committed.

rollback segment
A portion of the database that records actions that should be undone under certain circumstances. Rollback segments are used to provide transaction rollback, read consistency, and recovery.

S

sar
A command that is associated with the system activity report package.

screen pop
A method of delivering a screen of information to a telephone operator at the same time a telephone call is delivered. This is accomplished by a complex chain of tasks that include identifying the calling party number, using that information to access a local or remote ORACLE database, and pulling a “form” full of information from the database using an ORACLE database utility package.

script
The set of instructions for the Intuity CONVERSANT VIS to follow during a transaction.

Script Builder
An optional software package that provides a menu-oriented interface designed to assist in the development of custom voice response applications on the VIS.

SCSI
See “Small Computer System Interface.”

shared database table
A database table that is used in more than one application.

shared speech
Speech that is a part of more than one application.

shared speech pools
A parameter that allows the user of a voice application to share speech components with other applications.

Single Inline Memory Modules
A method of containing random access memory (RAM) chips on narrow circuit card strips that attach directly to sockets on the CPU circuit card. Multiple SIMMs are sometimes installed on a single CPU circuit card.

single-threaded application
An application that runs on a single voice channel.

slave
A circuit card that depends on the TDM bus for clock information.

Small Computer System Interface
A disk drive control technology in which a single SCSI adapter card plugged into a PC slot is capable of controlling as many as seven different hard disks, optical disks, tape drives, etc.

software
The set or sets of programs that instruct the computer hardware to perform a task or series of tasks — for example, UnixWare software and the Intuity CONVERSANT VIS Version 5.0 software.

software upgrade
The installation of a new version of software. The existing platform and circuit cards are kept.

source system
The system from which you are upgrading (that is, your system as it exists *before* you upgrade).

speech energy
The amount of energy in an audio signal. Literally translated, it is the output level of the sound in every phonetic utterance.

speech envelope
The linear representation of voltage on a line. It reflects the sound wave amplitude at different intervals of time. This envelope can be plotted on a graph to represent the oscillation of an audio signal between the positive and negative extremes.

speech file
A file containing an encoded speech phrase.

speech filesystem
A collection of several talkfiles. The filesystem is organized into 16-Kbyte blocks for efficient management and retrieval of talkfiles. The Intuity CONVERSANT VIS speech filesystem is not consistent with standard UNIX filesystems, and can not be referenced with standard UNIX commands such as **ls**, **cat**, etc.

speech modeling
Creating WholeWord speech recognition algorithms by collecting thousands of different speech samples of a single word and comparing them all to obtain a statistical average of the word. This average is then used by a WholeWord speech recognition program to recognize a single spoken word.

speech phrase
A continuous speech segment encoded into a digital string.

speech space
An area that contains all digitized speech used for playback in the applications loaded on the system.

standard speech
The speech package containing simple words and phrases produced by AT&T for use with an Intuity CONVERSANT VIS. This package includes digits, numbers, days of the week, and months, each spoken with initial, medial, and falling inflection. The speech is in digitized files stored on the hard disk to be used in the voice prompts played by the VIS.

standard vocabulary
A standard package of simple word speech models provided by AT&T and used for WholeWord speech recognition purposes. These phrases include the digits "zero" through "nine," "yes," "no," and "oh."

string
A contiguous sequence of characters treated as a unit. Strings are normally bounded by white spaces, tabs, or a character designated as a separator. A string value is a specified group of characters symbolized by a variable.

Structured Query Language
A standard data programming language used with data storage and data query applications.

subword technology
A method of speech recognition that recognizes phonemes or parts of words of American English vocabularies. See "whole-word technology."

switch
A software and hardware device that controls and directs voice and data traffic. A customer-based switch is known as a "private branch exchange."

switch hook
The device at the top of most telephones that is depressed when the handset is resting in the cradle (on hook). The device is raised when the handset is picked up (the telephone is off hook).

switch hook flash
A signaling technique in which the signal is originated by momentarily depressing the "switch hook."

switch interface administration
The component of the VIS that enables you to define the interaction between the VIS and switches by allowing you to establish and modify switch interface parameters and protocol options for both analog and digital interfaces.

switch network
Two or more interconnected switching systems.

synchronous communication
A method of data transmission in which bits or characters are sent at regular time intervals, rather than being spaced by start and stop bits. See also "asynchronous communication."

System 75
An advanced digital switch supporting up to 800 lines that provides voice and data communications for its users.

System 85
An advanced digital switch supporting up to 3000 lines that provides voice and data communications for its users.

system administrator
The person assigned the responsibility of monitoring all VIS software processing, performing daily system operations and preventive maintenance, and troubleshooting errors as required.

system architecture
The manner in which the Intuity CONVERSANT VIS software is structured.

system message
An event or alarm generated by either a VIS or end-user process.

system monitor
A component of the VIS in which tests are performed to verify that each incoming telephone line and its associated tip/ring or T1 card is functional. Through the "System Monitor" component, you are able to see displays of the Voice Channel and Host Session Monitors.

T

T1
A digital transmission link with a capacity of 1.544 Mbps.

table
A collection of records that are logically grouped together.

talkfile
An ASCII file that contains the speech phrase tags and phrase tag numbers for all the phrases of a specific application. The speech phrases are organized and stored in groups. Each talkfile can contain up to 65,535 phrases and the speech filesystem can contain multiple talkfiles.

target system
The system to which you are upgrading (that is, your system as you expect it to exist *after* you upgrade).

TDM
See "time-division multiplex."

telephone network connection
The point at which a telephone network connection terminates on an Intuity CONVERSANT VIS. Supported telephone connections are Tip/Ring and T1.

Terminal Emulator
Software that allows the VIS to temporarily transform itself into a "look alike" of an IBM 3270 terminal. In addition to providing full 3270 functionality, the Terminal Emulator enables you to transfer files to and from UNIX.

Text-to-Speech
An optional feature that allows an application to play speech directly from ASCII text by converting that text to synthesized speech. The text can be used for prompts or for text retrieved from a database or host, and can be spoken in an application with prerecorded speech. Text-to-Speech application development is supported through Script Builder.

ThickNet
A 10-millimeter (10BASE5) coaxial cable used to provide InterLAN communications.

ThinNet
A 5-millimeter (10BASE2) coaxial cable used to provide InterLAN communications.

time-division multiplex
A method of serving a number of simultaneous channels over a common transmission path by assigning the transmission path sequentially to the channels, with each assignment being for a discrete time interval.

Tip/Ring
A term used to denote analog telecommunications using four-wire media.

Token/Ring
A ring type of local area network that allows any station in the network to communicate with any other station.

trace
A command that can be used to monitor the execution of a script.

traffic
The flow of information or messages through a communications network for voice, data, or audio services.

transaction
Comprised of the exchanges between the caller and the voice system. A transaction can involve one or more telephone network connections and voice responses from the Intuity CONVERSANT VIS. It can also involve one or more of the VIS optional features, such as speech recognition, 3270 host interface, FAX response, etc.

transaction state machine process
A multi-channel IRAPI application that runs applications driven by script information.

transient process
A process that is created dynamically only when needed.

troubleshoot
The process of locating and correcting errors in computer programs. This process is also referred to as debugging.

TSM
See “transaction state machine process.”

TTS
See “Text-to-Speech.”

U

UNIX Operating System
A multiuser, multitasking computer operating system developed by the Bell Telephone Laboratories division of AT&T.

UNIX shell
The command language that provides a user interface to the UNIX operating system.

upgrade image tape
A tape, optionally provided to you by the Technical Service Organization, containing the new operating system and Intuity CONVERSANT VIS V5.0 base software in a standard configuration which is compatible with your target system.

upgrade scenario
The particular combination of current hardware, software, application and target hardware, software, applications, etc.

V

vi editor
A screen editor used by the Intuity CONVERSANT VIS to create and change electronic files.

virtual channel
A channel that is not associated with an interface to the telephone network (Tip/Ring, T1, or PRI). Virtual channels are intended to run "data only" applications which do not interact with callers but may interact with DIPs. Voice or network functions (for example, coding or playing speech, call answer, origination, or transfer) will not work on a virtual channel. Virtual channel applications may be initiated only by a "virtual seizure" request to TSM from a DIP.

VIS
See "Voice Information System."

vocabulary
A collection of words that a VIS is able to recognize using either WholeWord or FlexWord speech recognition.

vocabulary activation
The set of active vocabularies that define the words and wordlists known to the FlexWord recognizer.

vocabulary loading
The process of copying the vocabulary from the system where it was developed and adding it to the target system.

voice channel
A channel that is associated with an interface to the telephone network (Tip/Ring, T1, or PRI). Any Intuity CONVERSANT VIS application can run on a voice channel. Voice channel applications may be initiated by being assigned to particular voice channels or dialed numbers to handle incoming calls or by a "soft seizure" request to TSM from a data interface process (DIP) or the **soft_szr** command.

Voice Information System
A computer connected to a telephone network that handles touch-tone input, voice response, and line transfer. The Voice Information System uses a screen-based, menu-driven user interface to interact with the system operator or administrator.

voice processing co-marketer
A company licensed to purchase voice processing equipment, such as the Intuity CONVERSANT VIS, to market and sell based on their own marketing strategies.

voice response output process
A software process that transfers digitized speech between system hardware (for example, Tip/Ring and SP cards) and data storage devices (that is, hard disk, etc.)

Voice System Administration
The means by which you are able to administer both voice- and nonvoice-related aspects of the system.

VROP
See “voice response output process.”

W

warning
An admonishment used when there is a possibility of equipment damage.

WholeWord speech recognition
An optional feature based on whole-word technology that provides speaker independence, connected digit recognition, key word spotting, prompt interrupt, and DTMF support functionality. See “whole-word technology.”

whole-word technology
The ability to recognize an entire word, not the phoneme or a part of a word. See “subword technology.”

wink signal
An interruption of current to a busy lamp indicating that there is a line on hold.

word
A unique utterance understood by the recognizer.

wordlist
A set of words identified by a wordlist name. If the wordlist is part of an active vocabulary, the wordlist name appears as a recognition type in the Prompt & Collect mode field.

word spotting
The ability to search past extraneous speech during a recognition.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z