introducing

A UNIX Based Telephone Speech Data Collection System

prepared for:

Linguistic Data Consortium
441 Williams Hall
University of Pennsylvania
Philadelphia, PA 19104-6305

by:

Richard J. Duncan, Joseph Picone
Institute for Signal and Information Processing
Department of Electrical and Computer Engineering
Mississippi State University
Box 9571
413 Simrall, Hardy Rd.
Mississippi State, Mississippi 39762
Tel: 601-325-3149
Fax: 601-325-3149
email: duncan@isip.msstate.edu

EXECUTIVE SUMMARY

In the last five years, a direct digital interface to the telephone network has become the standard method of speech data collection for large speech corpora involving telecommunications applications. In particular, the T1 interface is popular because it is a cost-effective way to deliver a large number of voice channels. Unfortunately, such systems, most often based on PC hardware, use closed architectures consisting of proprietary software and hardware designs, resulting in a strong dependence on custom software from a single vendor. Vendors have repeatedly demonstrated an inability to deliver timely and cost-effective solutions for speech research, resulting in a great deal of wasted time and money, with no industry-standard solution in sight.

The Institute for Signal and Information Processing (ISIP) introduces a Unix-based platform for speech data collection based on an open-architecture design. There exist two reasonable alternatives for digital interfaces in modern day telephony: T1 and ISDN. T1 is currently more pervasive and more cost-effective when a great number of telephone lines are to be used in a large scale data collection application (typically the break point is between 4 and 8 lines). The platform uses a very inexpensive and popular workstation, a Sun Sparcstation 5, as its host.

T1 interfaces for Sun workstations are hard to find. The market leader in this niche arena is a system developed by Linkon Corporation. This system includes a T1 communications card that occupies one SBus slot and performs all data transmission functions, and a two-slot multi-DSP module that handles all call processing. The Linkon board is accompanied by some low-level, general-purpose software which serves as the foundation for building telecommunication applications. We have developed a hierarchy of software libraries extending the Linkon system to elegantly perform the interactions specific to speech data collection. We provide templates for the most common types of data collection: prompt and record modalities required for corpora such as POLYPHONE, and two-sided conference-style recording required for SWITCHBOARD-type corpora. We also provide easy to use interactive tools to build such applications from scratch.

The development effort for this data collection system involved four components: the specification of an operational T1-based system, a C++ wrapper to vendor-specific hardware functions, a fully hierarchical C++ code-base to manage all aspects of a speech data collection system, and an intuitive graphical user interface (GUI) that allows rapid prototyping of data collection applications. We also conducted extensive testing on naive subjects to optimize the GUI so that minimal training is required to bring new engineers on-line with the system. The C++ code, adherent to ISIP's strict standards of object-oriented data-driven programs, provides an abstraction to the hardware. The high-level data collection software (including the GUI) is hardware independent, making the future transition to other hardware systems a viable proposition.

We have developed a fully-expandable, robust system for platform-independent collection of telephone speech data. Our object-oriented software libraries and easy-to-use GUI provide a powerful tool with which even a novice user can efficiently generate complex applications. This document provides a detailed overview of our system, as well as a tutorial to illustrate prototyping of applications. As the speech community concentrates more on SWITCHBOARD-type telephone data, this system will play a pivotal role in speech recognition research.

TABLE OF CONTENTS

1. ABSTRACT

It is highly desirable to collect speech data from the telephone network by digitally interfacing to the network. This avoids an additional A/D conversion normally required by analog telephone data collection hardware. There exist two reasonable alternatives for digital interfaces in modern day telephony: T1 and ISDN. T1 is currently more pervasive and more cost-effective when a great number of telephone lines are to be used in a large-scale data-collection application (typically the break point is between 4 and 8 lines). T1 interfaces for Sun workstations have become a necessity in the speech research community. The market leader in this niche arena is a system developed by Linkon Corporation.

Using the Linkon system, we have developed a fully-expandable, robust system for platform-independent collection of telephone speech data. Our object-oriented software libraries and intuitive GUI provide powerful tools with which even a novice user can efficiently prototype complex applications. Using the system one can generate programs which range from simple single-user prompt/record demonstrations to robust SWITCHBOARD-type multi-user applications. This document provides a detailed overview of our system, a complete programmer's reference guide, and a step-by-step tutorial to illustrate prototyping of applications.

2. HISTORICAL PERSPECTIVE

The first system featuring a digital interface that was deployed for full scale data collection in speech research was the T1-based system built on the Intervoice Robot Operator hardware platform. This environment featured an IBM PC with an interface consisting of two proprietary boards, an OS/2 operating system (in its most recent generation), and a 4GL programming language for rapid application prototyping. It was used to collect the SWITCHBOARD corpus, and was recently in use on the CALL HOME and Voice Across Hispanic America (VAHA) projects[1].

What was wrong with this system? First and foremost, it is a closed architecture based on a platform that is incompatible with those currently used in speech research. Hence, its acceptance by the community as a general purpose platform has been slow. Second, for most projects, extensive firmware modifications have been required by the vendor to perform a particular style of data collection. Such modifications have historically been very expensive, and have caused numerous delays to the projects requiring them. In short, the vendor has traditionally been very unresponsive to the need for such modifications. Third, the current platform requires a steep learning curve involving a nontrivial custom environment. Operators often have to be educated directly by the vendor, and require several months to come up to speed. The cost of maintaining and operating the system is extremely high.

At the time the previous system was developed, there were no alternatives. Recently, though, several vendors have announced similar capabilities on general purpose Unix workstations. For the first time, it is feasible to duplicate the functionality of the Intervoice system in a much less expensive Unix environment, built from standard programming tools well-known within the speech research community. The advantages of this system are obvious and, most importantly, will result in significant time and money savings for LDC. The virtues of the open architecture

will become apparent in time as we demonstrate an ability to solve problems without the burden of an unresponsive vendor.

XTL teleservices is an object-oriented C++ based platform developed by Sun Microsystems for desktop call-processing applications. XTL was designed to provide an application programmer's interface (API) for the development of desktop applications, transparent porting between analog, ISDN, and ATM based technologies, basic building blocks of call processing (e.g. DTMF and silence detection), and other specialized services such FAX, modem, and video capabilities. There are drawbacks to using Sun's XTL platform as the core of a telephony application system. First of all, XTL is not free software, it is licensed by host. Also, it relies on Sun tools, making portability a problem. The aforementioned drawbacks of XTL and Intervoice indicate the reasons we decided not to use it as part of our telephony environment.[1]

Platform independence was a major design goal for this project, so why not Java? Sun's Java webpage describes the Java Speech API, which will allow Java applications and applets to incorporate speech technology into user interfaces. It will leverage the audio capabilities of other Java Media APIs, and when combined with the Java Telephony API, will support advanced computer telephony integration [5]. All of this is, of course, stated in the futures tense because this integration is not complete at present. Java was still in its infancy at the beginning of this project. It is still an open issue whether such a language can support demanding real-time applications such as though described below. It would not have been possible to execute this project in Java at the time it was started.

Similar to XTL, our approach was to develop a system whose core code was written in C++, was object-oriented, and somewhat hardware-independent. The designers of the XTL system created a good general form of implementation for the system, so we adopted many of the basic premises that XTL uses (though you could argue these features are inherent in any X-based GUI programming tool). The advantages of our code over XTL is portability and less strict installation standards, as well as being available as unencumbered shareware. We have created a smaller, robust, more dedicated system to handle a smaller range of speech data collection applications more easily and efficiently.

1. It is interesting to note that at the start of this project in Spring'95, XTL seemed a viable approach. By the end of this project in Spring'98, XTL was no longer a feasable option.

APPLICATION BUILDER (GUI) 3

The easiest way to rapidly prototype a telephone speech data collection application is to use the graphically based application builder. **lk_appbuilder** is the highest level interface available for the system. It allows the user access to all possible parameters in all situations, thus providing a simple interface to data collection application development. On-line, context sensitive help is always available, detailing the specifics of each parameter. A novel feature to the system is the graphical application debug mode, which maps a running application to the screen in real time. This utility is optimized to have an easy learning curve for novice users, yet is powerful enough to be useful to experienced users. Hence **lk_appbuilder** is the preferred way to prototype real systems.

The Application Builder is located in **$LINKON/bin/$ISIP_BINARY/lk_appbuilder**.

3.1. Quick Start with the Application Builder

To create a data collection application, you can start by filling in the general parameters (configuration menu, see Figure 3.10). After these are set to your liking, you may proceed to insert items (item menu, see Figure 3.7) for the different operations. Please note that most of the parameters are not necessary for every system, so leaving a parameter blank (or unchanged from the default) will often be the desired choice. For most areas of the screen, context sensitive help can be found by clicking the right mouse button.

The easiest way to learn how to build applications is by looking at examples. Several example applications are included with the distribution to facilitate learning **lk_appbuilder**. These sample applications can be found in the **$LINKON/util/example_applications/** directory. Use the *Open* function under the *File* menu (Figure 3.9) to load these pre-written applications for viewing and editing.

The simplest full application that can be created through the system is a type I data collection system, in which a single user is prompted to speak utterances. An example of such a system may be viewed in **$LINKON/util/example_applications/v2.0/type_1/config/config_file_0.dcol**. We will step through the development of such a system in Section 3.4.

It is often desirable to provide the user with a menu and allow branching from the user's choice. A simple example of this implementation can be studied in the shell application located in **$LINKON/util/example_applications/v2.0/type_1/menu/menu0.dcol**.

The label type II is used to refer to a SWITCHBOARD type data collection system where conversational speech is recorded from two users simultaneously. A sample application is located in **$LINKON/util/example_applications/v2.0/type_2/config/config_file_0.dcol**. Also, Section 7 describes the implementation of the LDC SWB-III protocol in greater detail.

3.2. Basic Features

The goal in designing the application builder is to make it powerful enough to design any application the system itself can handle, but at the same time to keep it simple enough for anyone to use without much experience or having to read this entire document. Figure 3.1 shows the basic layout of the application builder's main window. Remember that the right mouse button offers context sensitive help in most situations.

The application being designed is displayed simultaneously in two forms. The top half of the screen shows the call flow in a graphical plot. Each item is represented by an icon. Abort flow and data flow between items (the most complicated part of the any design process) are represented by sets of arrows. The arrows atop the items represent the abort control flow— what happens when an item does not execute successfully. The arrows in the center represent normal control flow— the default operation. The arrows along the bottom show the way data is moved around through the application.

The second way the application is displayed is through text-based lists on the bottom half of the

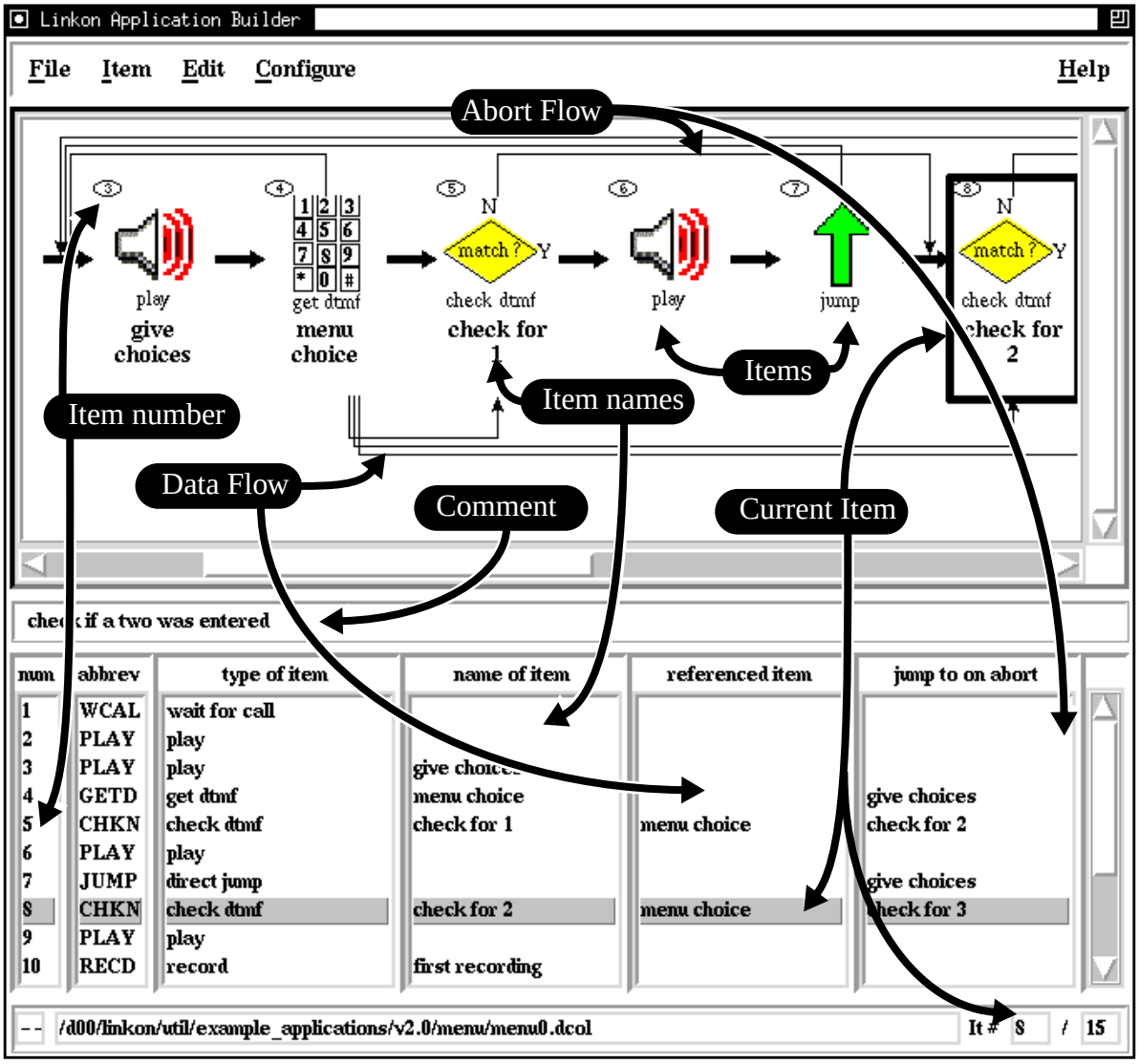


Figure 3.1: screen capture of **lk_appbuilder**'s main window

screen. Each item has a **number** (far left), an **abbreviation** for the type, and **type**. Each item may optionally have a **name of item, referenced item**, and/or **jump to on abort** specification. These parameters are displayed because they are closely interdependent upon other items. Item parameters are set through the item configuration window, which is activated by double clicking on either the item in the lists or the icon. When item configuration is selected, a window like the one shown in Figure 3.2 will pop up. Internal data is changed instantly when a value is changed in this window — hence there is no “ok” or “set” button. Notice that the parameters listed on the control screen are updated along with the text field. Often, the desired value for a parameter is already set as the default option. Simply leaving the value as is will configure the system to use this setting.

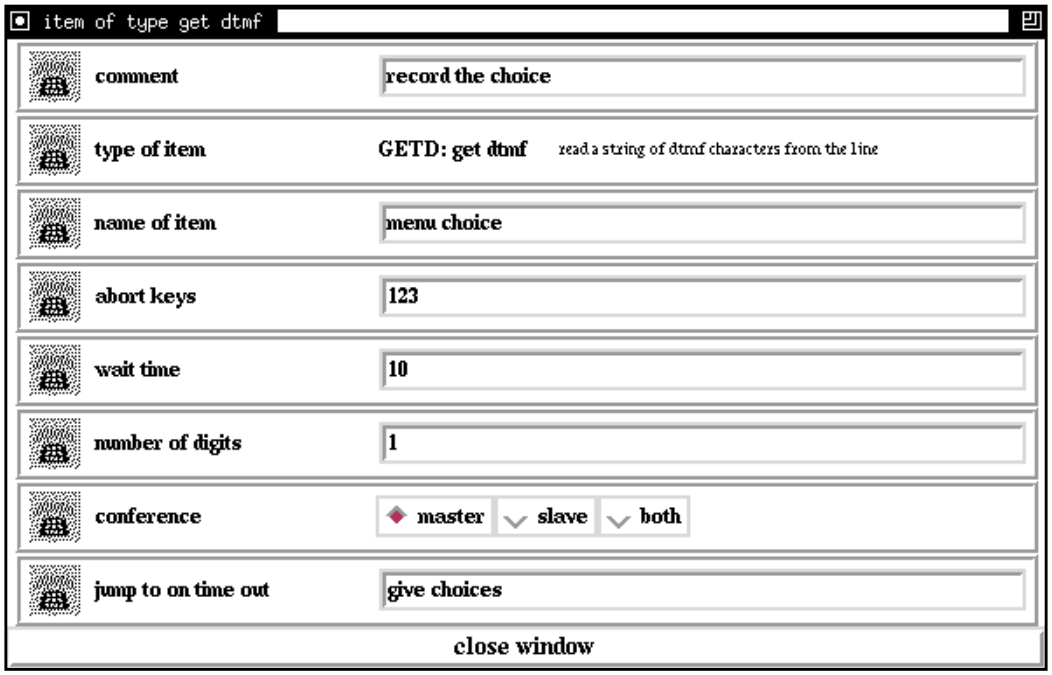


Figure 3.2: Example of the item configuration window

3.3. Advanced Features

The advanced features provide the experienced user with more power to quickly design applications. The advanced editing tools can be accessed through the options under the *Edit* menu (Figure 3.3).

Often when editing an application, it is desirable to cut an existing item and either move or copy it to another location. Both of these operations can be performed on the current item by using the *Cut* and *Copy* command under the *Edit* menu. After cutting the desired item, move to the location in the current application where you wish the item to be placed and use the *Paste* command. The item will be inserted in this location as if a new item is chosen from the *Item* menu, though it will retain all manual configuration options. If the item has a name attribute set, a string will automatically be appended to guaranty uniqueness.

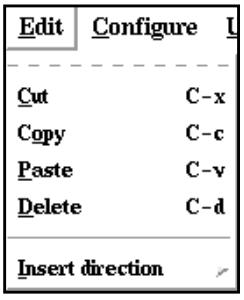


Figure 3.3: Edit menu

Insert direction allows you to change the insert behavior of the application builder. By default, new items (and pasted ones) are inserted after the current item. Changing this option will cause new (or pasted) items to be inserted before the current item. This is necessary if you wish to insert a new item before the first item in the system, but can be useful in other situations as well.

The bottom of the control window, shown in Figure 3.4, has an Emacs-like mode line. From this

Figure 3.4: The application builder's modeline.

Key	Function	Key	Function
Ctrl-q	quit	Ctrl-f	forward one item
Ctrl-z	suspend	Right	forward one item
Ctrl-o	open application	Ctrl-b	backward one item
Ctrl-s	save application	Left	backward one item
Ctrl-p	print application	Ctrl-F	forward 5 items
Ctrl-r	revert to saved	Ctrl-Right	forward 5 items
Ctrl-l	redraw plot	Ctrl-B	backward 5 items
Ctrl-c	copy item	Ctrl-Left	backward 5 items
Ctrl-d	kill item	Ctrl-a	just to first item
Ctrl-x	cut item	Ctrl-e	jump to last item
Ctrl-v	paste item	Ctrl-g	previous item
ESC	delete popup window	ESC	previous item

Figure 3.5: lk_appbuilder keybindings

window, you can monitor (from left to right) the save status, filename, current item number, and total number of items. The save status is very reminiscent of GNU Emacs. If the characters '- -' appear in this space, no changes have been made since the last save; while a '**' indicates that unsaved data exists. The filename appearing in the window is the most recent parameter filename. This file will be overwritten by selecting the *Save* command from the *File* menu. Use *Save As* to avoid overwriting or to save the file with another name. The 'current item number' shows the active item, and the 'number of items' field shows how many items are in the current application. Note that the number of items displayed is the actual number of items present in the system, not the estimated maximum number of items (the **num_items** parameter, described on page 67).

The application builder is equipped with several keyboard accelerators to speed up common tasks. The complete list of keyboard shortcuts is available as on-line help under the *Help* menu and is displayed in Figure 3.5. Another useful feature of the application builder is the graphical application debug mode, which is discussed at the end of this chapter.

3.4. Building a Simple Application

To run through the basic operation of the application builder, follow the steps described in this section. The application created will be a simple one — a system that first waits for a caller to call in, answers the phone, and plays a welcome message. The system will then prompt for the user to record an utterance, play the recorded utterance to the user, and ask if it is acceptable. The user will then respond with the touch tone keypad. Upon an acceptable utterance, the system plays a good-bye message and hangs up. This system can be visualized through the flowchart shown in

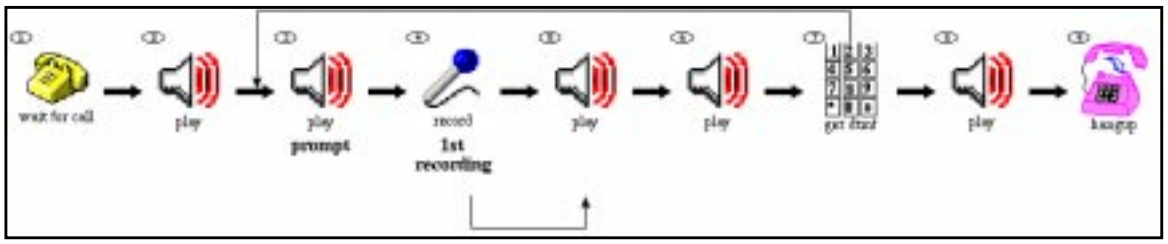


Figure 3.6: Application being designed

Figure 3.6.

To start, load the environment script with:

```
source $LINKON/LK_SETENV.sh;
```

Start the application builder by typing:

```
lk_appbuilder &;
```

The first thing this application should do is wait for a caller to phone in and when a call comes in, answer the phone. This is done by inserting a **wait for call** item; from the *Item* menu (Figure 3.6), select *WCAL: wait for call*. A configuration window like the one shown in Figure 3.8 will appear. There is no need to configure this item (the default values are sufficient), so click on the *close window* button for the **wait for call** configuration window.

Next, the application should play a welcome message to the user. This is programmed by inserting a **play** item (from the *Item* menu, select *PLAY: play*). Again, an item configuration window will pop up, but this time we need to set a value. Select the widget labeled *filename*. In this space, type the filename of a welcome message speech file. This is the only needed setting for this item, so click on the *close window* button.

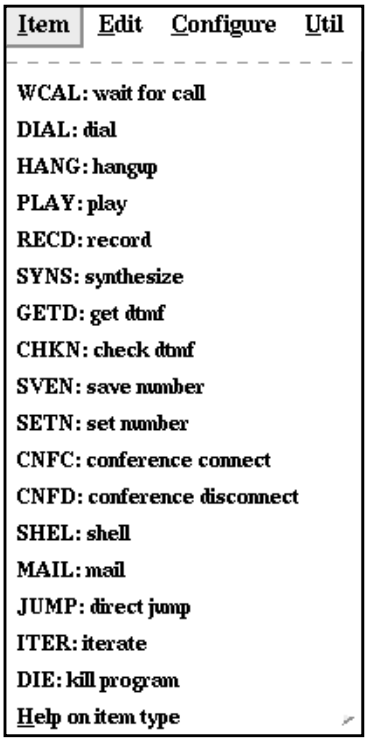


Figure 3.7: Item menu

Now we need to record a speech utterance from the caller, it is desirable to prompt the caller to speak by inserting another **play** item. This time, select the **filename** widget and enter a filename that contains an audio prompt (such as “Please start speaking now”) rather than a welcome message. Also, it is useful to name this item for later referral (we will jump back to this item if the recorded utterance is not acceptable), so enter the text “prompt” for the **name** of item.

In order to record audio data from the user, a **record** item will be used. This item type is accompanied by a large number of parameters; most of these are signal detection parameters (see Appendix D) that you should usually leave at the default values. Only two of these parameters need to be changed. The first is the **filename_template** parameter. Enter a value of “_r001” here—

this causes all files created by this item to be named in the form *[speaker_number]_r001_c00.sphere*. We also need to name the item so that the recorded utterance can be played back to the user. Enter the text “1st recording” for the **name** parameter.

If you now wish to verify that the recorded utterance is acceptable to the user, you should next play the recorded utterance back to the user. This is done by creating another **play** item. Instead of specifying a filename for this item (which would cause a static file to be played), we wish to playback the previously recorded file for this speaker. Setting the parameter **referenced item** to “1st recording”, the same name assigned to the record item, will facilitate this operation.

Figure 3.8: The wait for call configuration window

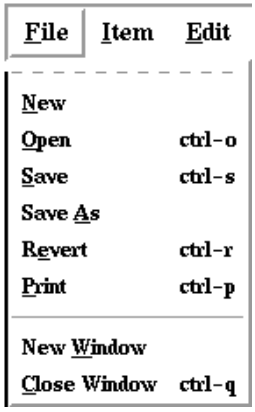


Figure 3.9: File menu

In order to have the user specify if the recorded utterance is acceptable or not, you need to have them respond with a dtmf key. First, you need a **play** item to play a prompt explaining this intent. This is implemented the same way as the first two **play** items. Next, you need to actually read the dtmf key from the keypad. This is done through the use of a **get dtmf** item. The configuration window for a **get dtmf** is shown in Figure 3.2. For our purposes, set the **abort_keys** parameter to all available touch tone keys, “0123456789*#.” This will cause any dtmf key event to stop execution of the item. Set the **wait_time** parameter to 10 seconds using the string value “10.0.” To configure the item to return to the prompt if no key is hit, enter “prompt” in the **jump to if timeout** parameter slot.

The last three items (**get dtmf**, **play**, and **hangup**) in the application assume that the user is satisfied with the recording. The **get dtmf** item keeps executing in a continuous loop until a verification key is hit. Another **play** item is inserted to thank the user for using the system and to indicate the end of the phone call — the mechanism for this is identical to the welcome **play** item described earlier. The last item in the list is **hangup** which terminates the call.

Now that the application’s individual items have been configured, select *General Parameters* from the *Configuration* menu (shown in Figure 3.10). Each of these general parameters needs to be configured to match the values listed in Figure 3.11. A description of each parameter can be obtained by clicking the right mouse button on the parameter, or by referring to Appendix B.

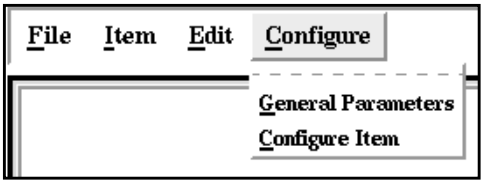


Figure 3.10: Configuration menu

The application is now complete. You will need to save this application to a file — under the *File* menu

parameter	value
max loop iterations	-1
database name	First Application
program name	First test program, run 1
administrative mail alias	<your email address>
speaker directory format	data/a%5.5d
needed disk space	0.25
data collection system	0
database logfile	first_logfile_0.text
progress file	first_progfile_0.text

Figure 3.11: Parameter settings

(Figure 3.9) select *Save As*. Use the dialog box to select a filename — “first_app.dcol” would be appropriate.

We have included a fully-written copy of this application in the system distribution. You may want to compare this example with your version. In order to do this, choose *New Window* from the *File* menu. In the new window, load the example application, **$LINKON/util/example_applications/v2.0/type_1/config/first_config.dcol**. Compare the two systems, and verify that they are identical.

Your new application should now be ready to run. Use the main application, **lk_driver** to run this. At the command line, type:

```
source $LINKON/LK_SETENV.sh
lk_driver -param first_app.dcol
```

Your new data collection application should now be running on the first channel, call in and see how it works.

3.5. Coordinating Multiple Items for Common Tasks

The true power of this speech data collection system is in making diverse items work together. Each of the items is designed to accomplish only a single, simple task. It is necessary to combine multiple items together, just as different statements are combined in a programming language. The very simple application described in Section 3.4 uses both data flow features (passing the newly recorded utterance for playback) and jumping (looping on the user’s key-press).

This section will describe a few of the most common blocks used in applications, such as looping structures, prompting for and reading dtmf (touch tone) keys from the user, menu-ing, and database access.

While a linear control flow may accomplish the most basic of speech-collection tasks, branching is a necessity for more sophisticated data-collection applications. Multiple forms of loop structures can be created through combining the **iterate** items with the **check dtmf** items. Such nested loop structures typically require the use of the **set number** item to reset internal counters for the inner loop. A basic counting loop is shown in Figure 3.12.

The **play** item loop is executed n times, where

$$n = \frac{check - initial}{step}$$

is the difference between the specific string set in the **check dtmf** item and the initial value set in the **iterate** item divided by the **iterate** item's **step** value.

If a pre-testing loop is desired, two **direct jump** items are necessary. An example of this loop structure is shown in Figure 3.13. Remember that an **iterate** item will increment itself the first time it is executed, so in this case the loop will execute

$$n = \frac{check - initial}{step} - 1$$

times. This structure, although more complicated, is generally more useful for data collection systems in that it will not execute an item if a precondition (such as a valid pin) is not met.

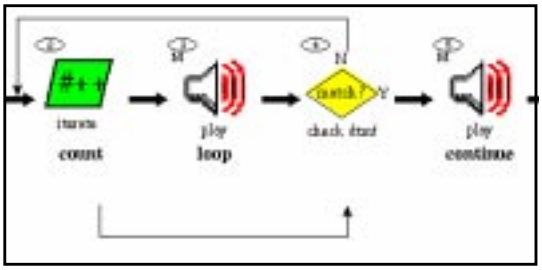


Figure 3.12: A basic counting loop

For a system as sophisticated as the full SWITCHBOARD protocol (which is described in detail on page 37), the importance of nested loops from a system design standpoint is clear. To convert a simple loop into a nested loop, a mechanism is required that allows arbitrary re-setting of an item's data. This is achieved with the **set number** item. It is used to alter the internal data of another item. To implement a nested loop as shown in Figure 3.14, a **set number** item is placed outside the inner loop to reset **count 1**. Items 5, 6, and 7 make up the inner loop here, with **count 1** acting as the LCV (loop control variable). The outer loop contains items 2, 3, 4, 8, and all the items in the inner loop, with **count 0** acting as its LCV.

An obvious question now arises — what would happen if the **set number** item (4) is placed inside the inner loop? The answer is consistent with any programming language, an infinite loop will occur. An infinite loop may also occur if

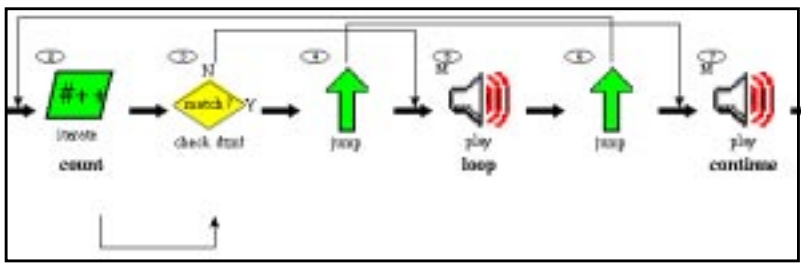


Figure 3.13: counting loop with prechecking of the LCV

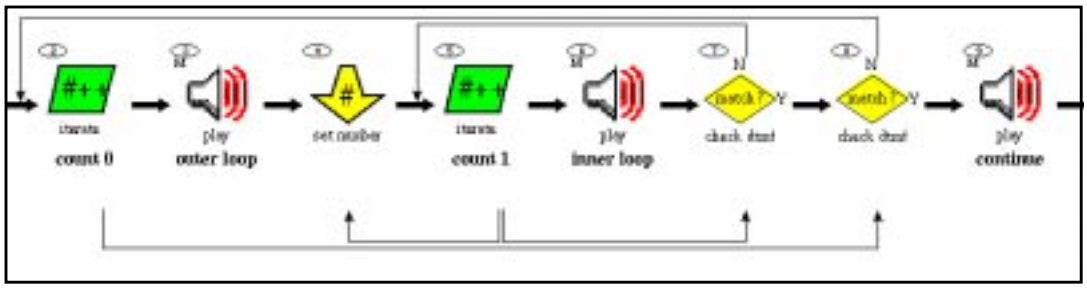


Figure 3.14: Nested counting loops

the step value of an **iterate** item LCV is set to 0 or the ranges are not correct. The control items are set up to give all power to the user, so caution is advised while placing and configuring these items.

Due to hardware constraints, the dtmf key repeat speed is not as fast during a **play** item as it is during a **get dtmf** item. This is the only reason that both items exist instead of a single, multipurpose item. In most cases where dtmf keys are read from the user, it is desirable for the user to be able to interrupt the prompt with data (key-presses). Once the user begins to enter a dtmf string in a **play** item, the **play** item will continue to read until the completion of user input (see page 49 for more information). If an unlinked **get dtmf** item was placed next in the call flow, the application would wait twice for the same information. Also, future items that need to reference this information would be easily confused as it would get stored in two different locations. To solve these problems, set the **short circuit dtmf** parameter in the **get dtmf** item to the name of the **play** item.

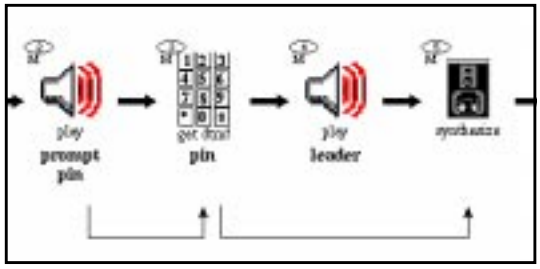


Figure 3.15: basic prompt dtmf read

Figure 3.15 shows an example of a prompted dtmf read. If the user begins to enter the pin during the prompt, the play item (**prompt pin**) will wait for the completion of the string. The **pin** item will not do anything with the user in this case, as a number value will be present in the **short circuit dtmf** parameter. Instead of waiting for a dtmf string, the **pin** item will simply copy the string contained in **prompt pin** as its own. Thus when item 5 references **pin** for synthesis, it will receive the number entered during **prompt pin**. If the user had not begun hitting dtmf keys during the prompt, **prompt pin**'s data will be null and thus not be imported by **pin**, and therefore the **pin** item will read the number and behave accordingly; and the referencing will behave normally.

In order to place multiple applications on the same phone line, data collection systems employ menus. There is no specific menu option available in the application builder's arsenal, but menus can easily be constructed with multiple **check dtmf** items. Figure 3.16 shows implementation of a very simple menu. After a prompt, the user enters a dtmf key. If '1' is entered, the first block (**play**ing instructions) is executed and it loops back to the prompt. If '2' is selected, then a prompt **record** block will be executed and it will end the call. On a user input of '3' the system will just end the call.

For more sophisticated data collection, it will probably be desirable to connect to an external database server (such as Oracle). For instance, the SWITCHBOARD protocol we have implemented on the Linkon system connects across the network to an Oracle server at LDC. Rather than create a specific item type for this task, the **shell command** item can be used. Perl scripts have been provided to handle all remote database functions through network sockets.

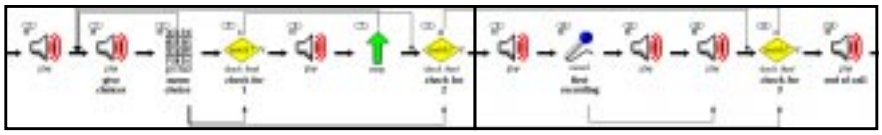


Figure 3.16: Implementing a menu

These perl scripts interface with the Linkon system through stdin and stdout, coordinating with various **shell command** items. The specifics of this interface are described in Section 7.

An example (Figure 3.17) illustrates how to glue all these features together. Here, the pin reading block of the SWITCHBOARD application is shown. Note that the block incorporates a prompted

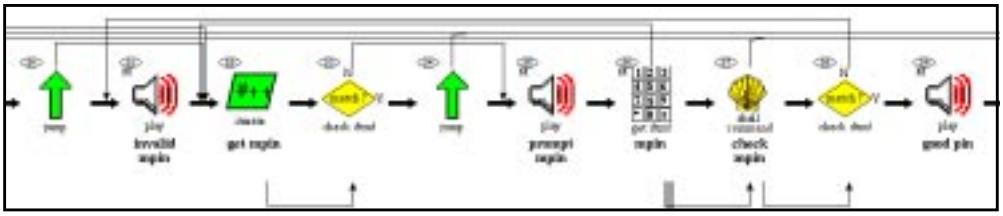


Figure 3.17: Reading and testing a user's pin

get dtmf, looping, and database access. First, the system prompts and reads the pin from the user. The pin is then passed to a **check mpin** item, which queries the database for the pin's validity. The **check dtmf** item (28) tests the output of **check pin**, it does not check the pin itself. If the pin is not valid, the user will hear an audible error message and get a few more attempts to enter a valid pin (through a pre-testing counting loop). This example shows how these simple items can be combined to form a very sophisticated block of code.

3.6. Using Progress Mode to Debug Applications

The approach towards building of speech collection applications on this system is quite similar to writing a computer program in a high-level language. The designer of the data collection application will run into many of the same problems that a programmer does, namely logical bugs. For simple applications with a strictly linear control flow, debugging is a simple process of using the application and noticing how at some point it diverges from the expected behavior. For a more complicated control flow, however, this quickly becomes unmanageable. For more serious debugging, the application can be run in the highest debug mode (`lk_driver -debug 3`). The driver program will output (to stdout) all information in this case, including specific calls to the Linkon hardware.

Figure 3.18: Util menu

The output from the debug mode is very complete, but is not very concise. If only the application itself needs to be debugged, we suggest using the progress mode. This mode will graphically display the running application. To run the application in progress mode, select *Run Debugger* from the *Util* menu (Figure 3.18). The debugging console (shown in Figure 3.19) will pop up directly to the right of the application builder's main window. If the application is running, the item selection square in the main window will automatically move to synchronously highlight the current active item.

The debugging console also provides a convenient way to monitor the string data contained in each item. The two columns on the bottom half of the debugging console are synchronized to the lists in the main window. The two new fields are *current DTMF* and *referenced DTMF*. The first field displays the data string contained in the item, the latter displays the data contained in the referenced item. For example, consider the case of the following two items: **get dtmf** and **check dtmf**. After the **get dtmf** item executes, its current DTMF field will be set to what the user enters on the touch-tone keypad. When the **check dtmf** item executes, it will load the same number as the referenced dtmf field. If a single specific key/number is used (instead of a list of numbers, as described on page 55), a quick visual inspection will indicate whether the test is successful.

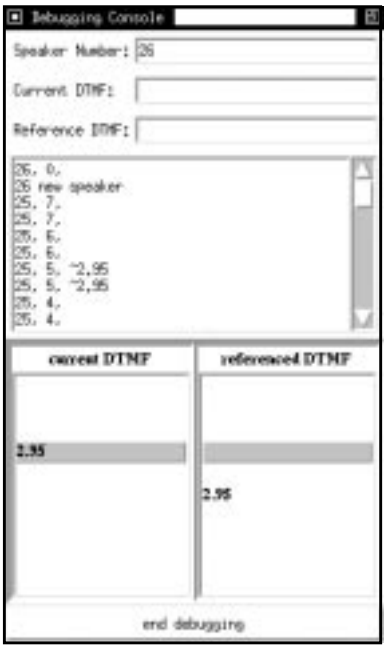


Figure 3.19: Debugging console

This chapter provided an introduction to the top-level user interface — the application builder; and described how to create simple applications through numerous illustrations. The appendices and on-line help contain the specific reference material necessary for real system development. Next, we discuss parameter files, the output from the application builder and the input to the real system.

PARAMETER FILES 4

The Application Builder, for the most part, simply allows the user a graphical interface with which to create parameter files for the system. These files can also be manually edited in any text editor. These parameter files contain all information necessary to run a data collection application, from system configuration to call flow specification. The specifications of these files are described in this chapter. Any parameter file can be edited by the application builder, manually (in a text editor), or by using a combination of the two methods. The only risk in moving from a manually edited file back to graphical builder is that the application builder is more stringent on comments. All comments except single lines beginning with the special '#*' sequence, located before each item and the general parameters, will be ignored (and lost) by the application builder. Other than the slight limitation on comment structure, advanced users may efficiently update scripts more efficiently with more powerful text processing tools (Perl, Emacs, Sed & Awk, etc.)

4.1. Quick Start

The fastest way to learn how to program application scripts directly is to look at existing applications. Either look at the code distributed with the software or the code written through the application builder. There are three example scripts included in the distribution which will facilitate a new user's instruction on parameter files.

A type I data collection system is characterized by the system prompting the caller to speak utterances. This is about the simplest application that can be created through the system. Look to $LINKON/util/example_applications/v2.0/type_1/config/config_file_0.dcol for the source.

It is often desired to provide the user a menu and allow branching from the caller's choice. A simple example of how this popular feature can be implemented can be viewed in $LINKON/util/example_applications/v2.0/type_1/menu/menu0.dcol

A type II data collection system is characterized by conversational speech recorded from two callers in conversation simultaneously. A sample application of this form of data collection is located in $LINKON/util/example_applications/v2.0/type_2/config/config_file_0.dcol.

4.2. Overview of the Scripting Language Syntax

The language in which the scripting (or parameter) files are written is similar to PERL. A specific Backaus-Naur style language definition is given in Figure 4.1. The main purpose of the scripts is to assign values to parameters, so the basic structure is:

```
parameter = value;
```

The *parameter* is any predefined attribute relevant to the current item, and the *value* is any string terminated by a semicolon. The parameter file consists of a magic string, general system parameters, and a list of item definitions. The general parameters are described in Appendix B. This being an object oriented system, each item in the list is a self executing module — the list is all that is necessary to specify a complete application flow. Descriptions of each of the item types can be found in Appendix A. Parameter descriptions can be found in Appendices C & D.

4.3. Referencing Data

As in most object oriented systems, user data is always kept WITHIN the individual items. The only two types of data that can be passed between objects are character strings and filenames. To access dynamic data that has been set earlier in the program, simply refer to the name of the item in which the data was collected. Each type of item interprets the data differently. See the item definitions (Appendix A) for specifics.

4.4. Control Flow

The normal control flow of a program is to traverse each item sequentially. Since this leads to rather restrictive and static systems, we provide capabilities for more complex flows. The flow

```
<file>::=MAGIC_STRING<comments><database info><comments><num items>
      <comments><items><comments>EOF
<database info>::=<comments><dbase statement> | <dbase statement><database info>
<dbase statement>::=<general param> = <value>; | <general param> = {<value>};
<comments>::=<comment> | <comments><comment> | ""
<comment>::= \n#<string>\n
<num items>::=num_items = <integer>;
<items>::=<comments>ITEM;<comments><item statements>
<item statements>::=<statements><type statement><statements>
<type statements>::=type = <item_type_name>; | type = <item_type_pnuem>;
<item_type_name>::=play | record | check dtmf | get dtmf | synthesize | wait for call
      | dial | hangup | mail | shell | conference connect | conference disconnect |
      direct jump | save number | set number | iterate | kill program
<item_type_pnuem>::=PLAY | RECD | CHKN | GETD | PLYN | WCAL | DIAL | HANG | MAIL | SHEL
      | CNFC | CNFD | JUMP | SVEN | SETN | ITER | DIE
<statements>::=<comments><statement> | <statement><statements>
<statement>::=<item param> = <value>; | <item param> = {<value>}; | ""
<general param>::= database_name | monitor_command | program_name | admin_alias |
      speaker_directory_format | disk_info | data_collection_system |
      teleconferencing_system | database_logfile | starting_speaker_number
<item param>::=<sigd param> | abort_keys | filename_template | conf_filename_template
      | wait_interval | filename_0 | filename_1 | filename_2 | speaker_mode | dtmf |
      name | ref_name | jump_name | address | shell_command | conf_channel | initial
      | step
<sigd param>::=sample_frequency | initial_pad | final_pad | max_initial_silence |
      max_final_silence | max_duration | min_energy
<value>::= <string> | <float> | <signed integer>
<string>::=<character> | <character><string>
<float>::=<signed integer>.<integer>
<signed integer>::=<sign><integer> | <integer>
<integer>::=<digit> | <digit><integer>
<sign>::= + | -
<digit>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<character>::=<lcase character> | <ucase character> | <ext character>
<lcase character>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<ucase character>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<ext character>::= | ' ' | '_' | '!' | '@' | '$' | '%' | '^' | '&' | '*' | '(' | ')'
      | '-' | '=' | '+' | '`' |  '~' | ':' | ''' | '<' | '>' | ',' | '.' | '?'
```

Figure 4.1: Parameter file language description — Backaus-Naur style

altering mechanism provided by the system can be simplified to *if ERROR jump NAME*, where *name* is the name parameter assigned to another item. This basic structure can be built upon to create very complex programs. Look at the *Control flow capabilities* section in each of the item definitions (Appendix A) for specific information on how different items can alter the flow.

4.5. System Log Files

A logfile is maintained by the system as specified by the **database logfile** parameter (page 68). This logfile is used to determine speaker numbers, hence multiple copies of the same system can be run to collect information to the same database simultaneously. The file locking system has been tested to work over NFS network mounted drives, so it is possible to run the same application on different machines, provided they all access the same logfile on one disk.

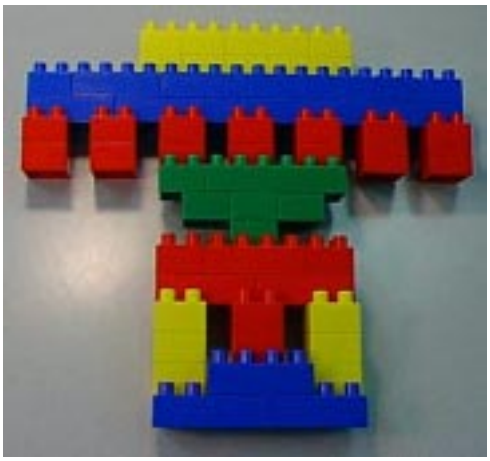
The logfile contains a single line entry for each caller. The first field in this line is the sequentially assigned speaker number. The second entry is a time stamp for the call. The third (and final) entry is a string with length equal to the **number of items** parameter. Each character in this string represents the state of each item. A blank space means that this item is in an intermediate state — it has yet to attempt execution. An ‘s’ indicates successful execution. An ‘h’ indicates that the user hung up on this item. An ‘x’ indicates that this item was not reached, something that will often happen if the user hangs up before the last of the messages is played.

The **number of items** parameter (page 67) exists mainly for the purpose of making logfiles usable after changes are made in the application. The logfile is read as a random access binary file, with field length being determined by the **number of items** parameter. Overestimating this parameter allows the same logfile to be used if minute changes are made in the configuration script.

An example logfile is shown in Figure 4.2. The first 21 speakers are collected using a uniform length configuration script, successful calls are logged for callers 1,3,5,6,8, and 10. The first eleven entries have 8 items, but space for 20 (null characters are represented by ^@). After these initial calls, the application was changed to have an additional item (speakers 50-55), but no calls were logged with this application. Callers 56 and 57 were back to 8 items, caller 58 was set up for a 5 item application. It is even possible that three different applications were running simultaneously using the same logfile if the **number of items** setting is identical for each script. Changing the number of items parameter and re-using an old logfile will generate a fatal error, possibly corrupting the logfile.

```
#0001 Sun Oct 26 16:45:13 1997 ssssssss^@^@^@^@^@^@^@^@^@^@
#0002 Sun Oct 26 16:45:13 1997         ^@^@^@^@^@^@^@^@^@^@
#0003 Sun Oct 26 16:48:27 1997 ssssssss^@^@^@^@^@^@^@^@^@^@
#0004 Sun Oct 26 16:48:27 1997         ^@^@^@^@^@^@^@^@^@^@
#0005 Sun Oct 26 16:51:28 1997 ssssssss^@^@^@^@^@^@^@^@^@^@
#0006 Sun Oct 26 16:52:44 1997 ssssssss^@^@^@^@^@^@^@^@^@^@
#0007 Sun Oct 26 16:52:44 1997         ^@^@^@^@^@^@^@^@^@^@
#0008 Sun Oct 26 16:59:01 1997 ssssssss^@^@^@^@^@^@^@^@^@^@
#0009 Sun Oct 26 16:59:01 1997         ^@^@^@^@^@^@^@^@^@^@
#0010 Sun Oct 26 17:04:33 1997 ssssssss^@^@^@^@^@^@^@^@^@^@
#0011 Sun Oct 26 17:04:33 1997         ^@^@^@^@^@^@^@^@^@^@
#0050 Fri Mar 13 15:32:49 1998          ^@^@^@^@^@^@^@^@^@
#0051 Fri Mar 13 15:36:26 1998          ^@^@^@^@^@^@^@^@^@
#0052 Fri Mar 13 15:48:52 1998          ^@^@^@^@^@^@^@^@^@
#0053 Fri Mar 13 15:50:05 1998          ^@^@^@^@^@^@^@^@^@
#0054 Fri Mar 13 15:50:50 1998          ^@^@^@^@^@^@^@^@^@
#0055 Fri Mar 13 15:52:43 1998          ^@^@^@^@^@^@^@^@^@
#0056 Fri Apr  3 02:11:06 1998         ^@^@^@^@^@^@^@^@^@^@
#0057 Fri Apr  3 02:12:57 1998         ^@^@^@^@^@^@^@^@^@^@
#0058 Fri Apr  3 02:28:58 1998      ^@^@^@^@^@^@^@^@^@^@^@^@^@
#0059 Fri Apr  3 02:30:31 1998         ^@^@^@^@^@^@^@^@^@^@
#0060 Fri Apr  3 02:34:44 1998         ^@^@^@^@^@^@^@^@^@^@
```

Figure 4.2: Example Log file



THE C++ INTERFACE 5

The linkon hardware was not specifically designed with speech data collection in mind, but rather to be a general purpose telecommunications board. The code developed for the system is built as a multi-level hierarchal C++ framework, which can be accessed from many different levels. The ***Data_collect***, ***Item***, and ***Item_list*** classes are highly parameter dependent; a safe design option is to leave these classes driven by the GUI and parameter file interface and avoid any hard-coded settings. The ***Tele_interf*** class is an appropriate level to interface with the hardware, allowing coding of a wide variety of applications without dealing directly with the device drivers. There is little advantage to going below the ***Tele_interf*** class for telephony board operations since most applications will use the functionality already built into this class. Additionally, all platforms supported by ISIP in the future will use the ***Tele_interf*** class as a common entry point, thus providing support for multiple platforms and compatibility with other systems.

5.1. System Architecture

Figure 5.1 provides an overview of the hierarchical software design. Building multiple layers into the system was necessary to provide simplified code design for future developers. ***Tele_interf*** is a generic computer/telephone interface, on top of which the parameter based system and GUI are built. This class is the break in the system architecture between hardware-specific and generic telephony operations, which guarantees portability of the high-level system to new hardware.

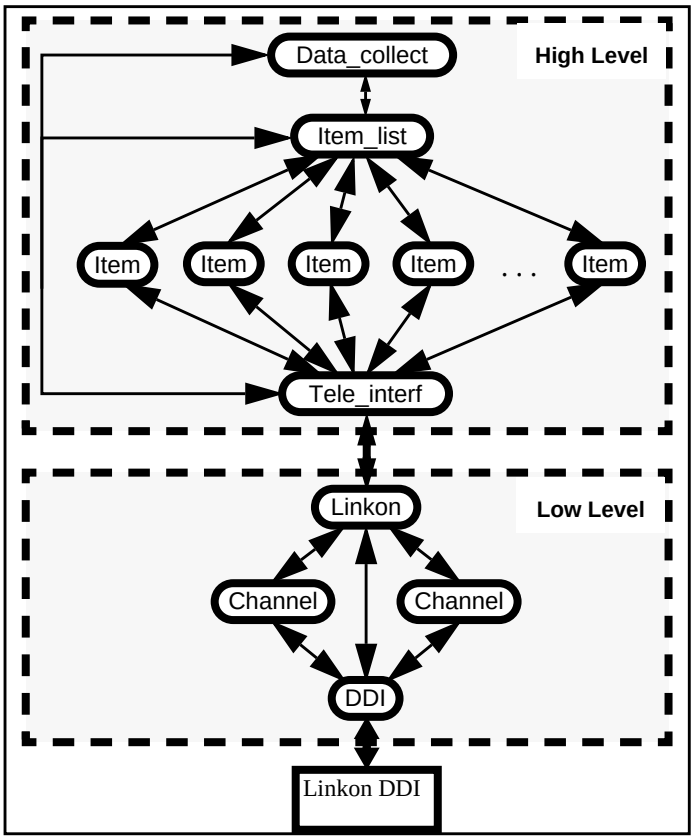


Figure 5.1: ISIP C++ code hierarchy

Three classes have been designed to interface directly with the Linkon fs4000 hardware: ***DDI***, ***Channel***, and ***Linkon***. The class at the lowest level is the ***DDI*** class, which acts as a wrapper for Linkon Corp.'s *Direct Driver Interface* functions [6]. A log can be kept of every call to the DDI library functions — a very useful feature in debugging hardware problems and communicating with the vendor. Each hardware line, or ***Channel***, is abstracted to an object, allowing most board functionality to be consistent for either party of a conferenced call. The ***Linkon*** class is used to coordinate and distribute the operations among channels and control teleconference connections.

The ***Tele_interf*** class abstracts the computer/telephone interface. There is no hardware-specific code in the ***Tele_interf*** class, so only code below this point in the hierarchy needs to be changed to adapt the system to hardware platforms other than Linkon. The majority of this section describes the ***Tele_interf*** class in great detail, as it is the C++ API to ISIP's abstract telephone object.

Items are self executing objects which are configurable to perform specific tasks. The most important parameter for an ***Item*** object is its **type** which defines how all other parameters are used. The different item types are described in detail in Appendix A. An ***Item_list*** is a collection of ***Item*** objects that describe the call flow in a complete data collection application.

The ***Data_collect*** class is used to keep track of every aspect of a data collection system. It configures the hardware (through a ***Tele_interf*** object), handles the log file (page 17), parses the parameter file for configuration, and executes the ***Item_list*** for each caller.

In addition to the classes described above that compose the hierarchy of hardware control and database collection, three classes (***Filename***, ***Signal_detector***, and ***Sample_comp***) are included, at

the lowest level, to assist in common transactions. The ***Filename*** class is designed to modularize the common operations of system filenames, from handling environmental variables to file locking. The ***Signal_detector*** class does nothing more than hold all of the signal detection parameters, but is far more convenient than passing multiple variables around. The ***Sample_comp*** class is used to perform audio format conversion and signal scaling.

5.2. Class Tele_interf — a Software Abstraction

Direct access to the board through C++ has been abstracted by use of the class ***Tele_interf***. Figure 5.3 shows the public functions available through this class which are relevant to telephone data collection. The purpose of this class is to abstract a generic computer/telephone interface whereby the system could easily be ported to other hardware with minimal modifications. Thus, there is no hardware-specific code (particularly no Linkon-specific code) in the ***Tele_interf*** class.

The remainder of this chapter will describe, in detail, the C++ interface to the hardware platform using ***Tele_interf***. While this chapter gives a high-level view of each of the functions, many of the specific details can only be observed by actually looking at source code. Thus, example code is included as the last section of this chapter (5.8) for the simple diagnostic programs described in the installation chapter. All code adheres to ISIP's strict standards of object-oriented data-driven programs, ensuring readability and simplicity. Since all of the source code is included with the distribution, a programmer may delve as deep into the system as they wish to determine the exact inner workings. However, this will not be necessary to design typical data collection applications.

5.3. Initialization and Configuration Methods

Any program that uses the system must construct a ***Tele_interf*** object —the constructor takes no arguments. Before performing any operations with the board, it is also necessary to initialize the ***Tele_interf*** object and instruct it on which channels to use. For a single channel application (such as Type 1 data collection), you have the option of either specifying a channel directly or allowing the system to choose from the bank of *free* channels. For a two channel application, both channels must be explicitly stated. The status of all channels is maintained in the $LINKON/proc directory

code	expansion	explanation
`f`	free	channel is free to be used by any application, even if not explicitly requested (sequential channel assignment).
`r`	reserved	channel is not in use, but may only be used by an application that specifically requests it (no sequential channel assignment).
`o`	open	channel is currently held by an application, with no current line activity.
`b`	busy	channel is held by an application and there is current line activity (connected to a user).

Figure 5.2: The four possible states of a channel.

```
// tele_interf: a class that provides a general high-level interface to
//              more specific telephone interface hardware classes
//
class Tele_interf{

public:
    // required methods
    //
    char_1* name_cc();
    volatile void error_handler_cc(char_1* method_name, char_1* message);

    // constructors/destructors
    //
    ~Tele_interf();
    Tele_interf();

    // initialization/configuration methods
    //
    logical_1 init_cc();                                    // 1 non-specific channel
    logical_1 init_cc(int_4 chan);                          // 1 specific channel
    logical_1 init_cc(int_4 master, int_4 slave);           // 2 specific channels

    logical_1 reset_cc();                                   // after each call

    logical_1 reset_conf_flag_cc();                         // for forced disconnect

    logical_1 select_cc(int_4 value);                       // select master/slave/both
    logical_1 select_cc(char_1* val);                       // select master/slave/both
    logical_1 configure_cc(char_1* keys);                   // configure channels
    logical_1 configure_cc();                               // configure channels

    int_4 get_chan_number_cc();                             // return channel number
    logical_1 get_con_status_cc();                          // remote connection ?

    logical_1 set_digits_cc(Filename num_file);             // set the digits

    // call management
    //
    logical_1 offhook_cc();                                 // basic call progress
    logical_1 onhook_cc();
    logical_1 hangup_cc();
    logical_1 dial_cc(char_1* tel_number);
    logical_1 dial_cc(int_4& status, char_1* tel_number);
    logical_1 wait_for_call_cc();
    logical_1 wait_for_call_cc(char_1 *ani);                // return ANI info
    logical_1 wait_for_event_cc(char_1* keys, float_4 timeout);

    logical_1 connect_cc();                                 // connect slave channel
    logical_1 disconnect_cc();                              // disconnet slave channel

    // method to set some data to be used in creating sphere files
    //
    logical_1 set_sphere_parameters_cc(char_1* database_name, int_4 speaker_number);
    logical_1 set_sphere_parameters_cc(int_4 speaker_number);
```

Figure 5.3: class Tele_interf public functions.

```
    // i/o methods, high level
    //
    logical_1 record_file_cc(float_4& rec_time, Filename& out_file,
              Signal_detector& sigd);
    logical_1 record_file_cc(float_4& rec_time, Filename& out_file,
              float_4 sample_freq, float_4 max_time);
    logical_1 play_file_cc(Filename& in_file);
    logical_1 quick_play_file_cc(Filename& in_file);

    logical_1 play_numbers_cc(char_1* numbers);
    logical_1 play_teli_number_cc(char_1* numbers);         // does pausing
    logical_1 get_dtmf_string_cc(char_1* numbers, char_1 abort, float_4 timeout,
              int_4 num_digits);
    logical_1 finish_dtmf_string_cc(char_1* numbers, char_1 abort, float_4 timeout,
              int_4 num_digits);

    logical_1 record_files_cc(float_4& rec_time, Filename& master_file,
              Filename& slave_file, float_4 sample_freq, float_4 max_time);
    logical_1 record_files_cc(float_4& rec_time, Filename& master_file,
              Filename& slave_file, Signal_detector& sigd);

    // i/o methods, low level
    //
    logical_1 start_read_cc(int_4& needed_size, Signal_detector& sigd);
    logical_1 start_read_cc();
    logical_1 start_write_cc();
    logical_1 stop_write_cc();

    int_4 read_cc(sample_type** obuffer, sample_type* buffer, int_4 buffer_size);
    int_4 read_cc(FILE* fp, int_4 max_size);

    int_4 read_cc(sample_type** obuf_m, sample_type** obuf_s, sample_type* buf_m,
              sample_type* buf_s, int_4 buffer_size);
    int_4 read_cc(FILE* master, FILE* slave, int_4 max_size);

    int_4 write_cc(sample_type* signal, int_4 nsamples);
    int_4 write_cc(FILE* fp, int_4 max_size);

    logical_1 generate_dtmf_cc(char_1 value);
    logical_1 get_dtmf_cc(char_1& dtmf, float_4 timeout);
    char_1 get_last_dtmf_cc();
};
```

Figure 5.3: class Tele_interf public functions.

in a file named channels.dat. The TCL application **lk_channel_handler** assists in monitoring and altering the channel states. The channels.dat file contains a single character entry for each physical channel on the board, with possible character codes ‘`f`’, ‘`r`’, ‘`o`’, and ‘`b`’. These codes are described in Figure 5.2.

To reflect the multiple methods available for channel selection, there exist three versions of the `Tele_interf::init_cc()` method. Calling `init_cc()` with no arguments will allocate the first *free* channel for use by your program. An overloaded version of init_cc() allows one to specify a channel to be used. In this case, the system will grab either a *free* or *reserved* channel.

Specifying the channel is advised to keep different applications on specific inbound telephone numbers — these channels should generally be reserved so they will not be arbitrarily grabbed by another program. Due to teleconferencing limitations on Linkon's hardware, a two line call requires adjacent channel numbers. In order to use two lines in the same application, the two-argument version of `init_cc()` must be used to specify both channel numbers.

The `get_channel_number_cc()` method will return an integer corresponding to the physical channel currently being used by your application (referred to as *active*), or `TELI_NO_CHAN` if no channel is set. `get_con_status_cc()` is used to query the status of a remote connection, returning `ISIP_FALSE` if no user is currently connected to the system on the active channel and `ISIP_TRUE` otherwise. The `configure_cc()` methods are used to reconfigure a channel to the default settings (a string of abort keys may also be included). The `reset_cc()` function is used to clear the abort status of the board between calls. `reset_conf_flag_cc()` resets the teleconferencing flag to the disconnected state —useful for a forced disconnect.

If your application requires multiple channels (such as SWITCHBOARD) you must specify the channel each method will act upon (i.e. the active *channel*). The `select_cc()` functions are used to set the active channel. The version which takes a character pointer accepts a string value: **master**, **slave**, or **both**. The codes **master** and **slave** set a single channel as active and this channel is used for performing all tasks until the next call to `select_cc()`. The codeword **both** indicates for the operation to be performed on both channels. Note that not all functions support multiple channels, and a few do not support single channels. For example, `dial_cc()` will only operate on a single channel, while the conference record method will only operate on both channels simultaneously. The conf_connect_cc() and conf_disconnect_cc() functions will ignore the active channel setting and always operate on the master channel.

The final configuration method is `set_digits_cc()`, which specifies the canned audio used by the `play_numbers_cc()` (**synthesize**) function. The argument to this function is a ***Filename*** object containing the name of a NIST sphere audio file. This file should contain the audio corresponding to '0'. The format of this filename should end in 0.ext (for example, $SWB_DCOL/prompts/say_0.sphere or 0.sp). The filename must have an extension. The '0' will be replaced with '1'-'9' and '11' to determine the rest of the digits, with '11' indicating a pause (specified by a ',' character in the number string). So, if you specify the filename to be "$PROMPTS/0.sphere," the files should appear on your system as $PROMPTS/0.sphere, $PROMPTS/1.sphere, . . . , $PROMPTS/9.sphere, and finally $PROMPTS/11.sphere.

5.4. Call management Methods

Once the board has been properly initialized, call management routines are used to connect with a user by beginning a call. The two ways to begin a call are to either wait for a user to dial into the system, `wait_for_call_cc()`, or to initiate an outbound call, `dial_cc()`. A user must be connected for any i/o methods to be performed. `hangup_cc()` disconnects a user from the system.

Most passive data collection applications will want to operate on inbound calls, with the users calling in at their convenience. `wait_for_call_cc()` is a blocking function which, once called,

does not return until a user has dialed in. This method may optionally return the Real-Time ANI information for the calling party (commonly referred to as caller-id). The ANI information returned will be in the form of a character string, with each character among those defined as valid dtmf codes (`TELI_TT_0` - `TELI_TT_9`, `TELI_TT_ABS`, `TELI_TT_LBS`, `TELI_TT_PAUSE`) in the header file *tele_interf_constants.h*. This method will always return `ISIP_TRUE`.

If your application needs to initiate an outbound call, use the `dial_cc()` methods. Both `dial_cc()` functions require that the telephone number to dial be specified as a character string and that each character in the string be a valid dtmf code. If a dial operation is successful, the function will return `ISIP_TRUE`, else it will return `ISIP_FALSE`. Success is defined in this context as the establishment of a connection to a live human. A status variable may be passed by reference to this method via the alternate overload to provide an integer code for the status of the dial. These codes are defined in *tele_interf_constants.h*. Support beyond the codes for human, ring no answer, and busy has not been extensively tested by ISIP.

The two hook functions, `onhook_cc()` and `offhook_cc()`, should not be used external to the ***Tele_interf*** class for typical applications. The higher level `dial_cc()`, `wait_for_call_cc()`, and `hangup_cc()` should be used instead. The hook functions are included in the interface only for completeness. ***Channels*** (or lines) are usually left in the offhook state so that the T1's hunt algorithm will not try to connect an inbound call to idle channels. The only channels that should be in the onhook state are those waiting for an inbound call and those performing a hook flash for the dial or hangup functions.

In a SWITCHBOARD-style application, the common flow is to wait for caller A to dial in and then dial out to caller B. Once two callers are on-line, the voice channels may be connected so that the callers can hear each other (as if caller A had directly dialed caller B). This is called a conference connection, or, as described by Linkon, Corp., a Linkon Expansion Bus (LEB) connection. The `connect_cc()` function is used to create a teleconferencing connection between the two channels; the `disconnect_cc()` function breaks the connection. Because only two channels may be connected in this manner, the Tele_interf class is designed to only operate on two channels. This design may be expanded in the future to support hardware advances.

With the Linkon hardware, it is not possible to write (play audio) to conference connected channels. If this is attempted, the connection will be broken and must be explicitly re-initiated by the program. This response was found to be less error prone than automatically re-initiating the connection after audio output.

5.5. Sphere Audio File Handling

The system is designed to handle high-level audio functions with NIST sphere files. A sphere file is an audio data file with an ascii header. The header is a block of name-value pairs, stored in a readable text format. The sphere file is prevalently used in the speech community to handle both waveform data and arbitrary information (such as database information). The format was developed by John Garafalo and Jon Fiscus at the National Institute of Standards and Technology Spoken Natural Language Processing group. Utilities and programming libraries for handling sphere files are available for download at the NIST website [7].

Since the system was designed to be used as a data collection platform, it includes some database information with every sphere file recorded. The `set_sphere_parameters_cc()` methods are used to set the database name and speaker number so as to include these fields in the sphere headers. Once set, the database name need not be set again, but the speaker number is usually changed for each new call. A summary of the sphere fields set is shown in Figure 5.4.

Although the Linkon hardware supports the μlaw audio data compression format, we observed errors when recording data in this mode (playback still worked fine). Most data would come off the T1 without problems, but the first packet (3504 bytes/samples, ~ 0.44 seconds of speech) would be corrupt if the caller spoke as the board went into record mode. If no energy was present on the line during this initial fraction of a second, no such errors were incurred. Linkon corporation has been advised of this error, but time will not be invested on our part to locate the source since there is an alternate method.

field	type	description
sample_count	int	number of samples in the file
sample_n_bytes	int	number of bytes per sample (1 for μlaw)
channel_count	int	number of channels in this file (always 1)
sample_rate	int	sample frequency of recording (always 8000)
sample_coding	str	format of the data (T1's use "μlaw" encoding)
sample_byte_format	str	format of each sample (always "1"). This parameter is used for multiple byte samples to specify byte encoding.
database_id	str	the database name (user specified)
speaker_number	int	the speaker number (user specified)
recording_date	str	unix date style string specifying the date of recording
recording_environment	str	always "telephone"
dcol_system	int	data collection system, or channel used, for this recording.
conf_system	int	the channel of the conferenced party, if applicable
sample_min	int	minimum sample value
sample_max	int	maximum sample value
sample_sig_bits	int	number of significant bits per sample

Figure 5.4: NIST sphere file header fields. The shaded entries are those standard to NIST sphere files, the rest are additional fields added by ISIP for database purposes.

Obviously, this is not acceptable for data collection purposes. This behavior does not present itself for linear data recording (16 bit signed linear samples), so the fix is to leave the board in linear configuration. The linear data is still perfectly quantized to 255 values of (directly corresponding to the sox and w_edit μlaw look up table after applying a scale factor of 2). Since the data is still 8 bit quantized in the linear format, the data transformation is perfectly invertible with the loss of no information.

The conversion is performed at the lowest level, a system designer deals only with μlaw data. Both direction of the conversion use look up tables to minimize CPU overhead. This is not the only binary operation performed on the data between disk files and the T1, it is also byte-swapped to accommodate the native Sun byte ordering (Big Endian). The design choice to do this processing online is to simplify the data collection process. Future hardware platforms will probably not have the same idiosyncrasies as the Linkon board, but will quite possibly present other challenges. Maintaining separate data sets for different hardware platforms would be a management nightmare. Since these byte operations require minimal CPU resources, these details are completely abstracted from the user.

The parameters described above represent the μlaw speech encoding used on T1 lines. Alternatively, the analog system uses 16 bit, uncompressed, single channel, raw signal data at a sample frequency of 8kHz. NIST sphere files for this system should be configured with the **sample_coding** set to “pcm” and the **sample_byte_format** set to “10.” The sample_max header field is used to normalize all linear signal data during playback. This play normalization feature is currently not available for the μlaw data format.

5.6. High-Level I/O Methods

High-level i/o methods are included in the ***Tele_interf*** class to simplify common operations. With these methods you can play an audio prompt, record the user’s speech, and read dtmf (touch tone) keys with little underlying knowledge of the hardware. Although multiple overloaded versions of these methods exist, it is not possible to predict every possible use of the board, so the low-level i/o methods used by these high-level functions are also included in this interface. A description of the low-level methods may be found in the next section (5.7).

The most basic form of data collection comes from the `Tele_interf::record_file_cc()` function. This records what the user speaks into the telephone microphone and saves it to a disk file. The interface to this function comes in two variants. One version uses the custom signal detector (page 77) to record a single speech utterance. The second version is set to record for a specified amount of time (in seconds). Both versions return in the `rec_time` parameter the length, in seconds, of the recorded audio file. The `record_file_cc()` methods can only be performed on a single channel at a time, and recording will stop if the user hangs up or presses any dtmf key.

To record data from both users simultaneously, use the `record_files_cc()` methods. These functions operate essentially the same way as the single channel record operation but with two files (one for each channel) specified rather than one. These files will contain perfectly

time-aligned data. The Linkon board returns interleaved buffers of data approximately 0.125 seconds long (this will likely vary for different hardware platforms). If necessary, one buffer of data will be cropped from one channel's data file to equalize recording lengths.

A second central feature of telephone interfaces is playing audio prompts to the user. One can provide this functionality by simply creating a ***Filename*** object pointing to an audio prompt file and calling `play_file_cc()`, passing this filename as the only argument. The `play_file_cc()` method will play the prompt to the user. These methods can be used to play a file to a single user or two users simultaneously.

The methods `play_file_cc()` and `quick_play_file_cc()` differ in that the former method will not return until the entire prompt has been played and the latter will return as soon as the audio data has been dumped into a buffer on the board. If the playback of multiple, uninterrupted prompts is desired, all but the last should be called with `quick_play_file_cc()`. Calling `play_file_cc()` with the last file will wait until all files (including the last) have finished playing before returning.

While the record method will be interrupted by any dtmf key, much more control is available for the play methods. Only keys configured to be in the abort mask, by calls to `configure_cc()`, will interrupt playback. Once an acceptable key is hit, there are two options for reading the value. First, you can call the low-level method `get_last_dtmf_cc()` to discover which single key was hit to cause the interrupt. For a more complex handling, you can immediately call the `finish_dtmf_string_cc()` method (operating much the same as `get_dtmf_string_cc()`, described below) to read a longer dtmf string from the board. The string returned by `finish_dtmf_string_cc()` will include, as the first character, the interrupt causing character hit during the play operation.

In order to provide easily processed user input, the `get_dtmf_string_cc()` function is designed to read a string of dtmf (touch tone) characters. This is an invaluable function for databases, nearly as useful as reading input from the keyboard is in text-based interfaces. This function is nice in that one function call will return the entire dtmf input (such as a PIN) from the user. `get_dtmf_string_cc()` reads data until an end of string event. This event may be generated in three ways. First, a time-out waiting for a dtmf key (each key-press resets the count) will terminate the read. Secondly, if the `num_digits` parameter is set, the read will terminate with the nth input character. Lastly, a terminal character can be specified through the `abort_key` parameter. This terminal character will not be returned as part of the string.

While the interface to the two functions, `get_dtmf_string_cc()` and `finish_dtmf_string_cc()`, is identical, the two methods behave slightly differently. The `num_digits` parameter specified in the call to `finish_dtmf_string_cc()` will include the first character which triggered the play operation to cease (passing a value of 1 will cause it to return immediately). Also, keys are read in much faster from the dedicated `get_dtmf_string_cc()` function due to hardware i/o limitations. So, if reading dtmf input through a prompt is not quick enough (i.e., it seems to drop characters), consider using the dedicated method.

original number	delimited number
8335	8335
3258335	325-8335
16013258335	1-601-325-8335
916013258335	9-1-601-325-8335
03258335	0-325-8335
98335	9-8335

Figure 5.5: Adding pauses to a telephone number

A set of high-level methods was also designed to synthesize number strings into speech for the user. Canned speech files for each number must be specified (see the `set_digits_cc()` method in Section 5.3). All characters will be played using the `quick_play_file_cc()` method to minimize pauses between characters, but the system will wait until playback of all numbers completes before returning from the function. Note that non-numeric characters in the data string will be ignored.

If the number to be synthesized is a telephone number, the speech signal can be delimited by appropriate pauses to maximize understandability. Use the `play_teli_number_cc()` method to automatically add these pauses to the speech signal. As it is difficult to describe exactly where these pauses are added, Figure 5.5 shows examples of the most common cases. Pauses will be added by inserting a pause character into the dtmf string (the ‘,’ character, as defined in `tele_interf_constants.h`). This file should be associated with the _11 digit, as described by `set_digits_cc()` in Section 5.3.

5.7. Low-Level I/O Methods

The high-level methods described in the previous section exist to simplify applications, but we cannot predict every possible use of the board. For this reason, the low-level basic methods are also at the programmers disposal. Through the use of these functions most any operation can be performed.

The read and write functions are with reference to the telephone line, not the disk. So, the read functions will record data coming from the telephone line into a disk file or memory buffer, the write functions will play data from a disk file or memory buffer to the telephone line.

The first step in recording data is to call a `start_read_cc()` method to configure the board to begin recording data. Two variants of this method exist. The first method is used if you desire to use the signal detector. In this form you pass a signal detector object (parameters are described in Appendix D). The method will return the size of the memory buffer required to record such an utterance as `needed_size`. The second variant, which takes no arguments, is used if no signal detection is necessary.

To record data with a signal detector, one must allocate a memory buffer large enough for `needed_size` samples to be stored since this `read_cc()` function will operate entirely in memory. `read_cc()` will set `obuffer` to hold a pointer to data within the allocated memory buffer. The position of `obuffer` indicates the start of the valid utterance. The `read_cc()` function will also return the number of samples of this valid utterance. Since no memory allocation is performed by the function, only the originally allocated buffer needs to be deleted after the operation is complete (and data saved).

If no signal detection is needed, the process is much simpler. The programmer needs only to specify an open file pointer for writing samples and the maximum number of samples to read. The user may prematurely abort either version of the read operation by hitting a dtmf key.

Since the `read_cc()` operations described can only be performed on one channel at a time, overloads exist to read both channels simultaneously in both modes. Simply pass two sets of memory pointers or two open file pointers to read from both channels. These dual channel read operations will return perfectly time aligned data. The Linkon board returns interleaved buffers of data approximately 0.125 seconds long (this will likely vary for different hardware platforms). If necessary, one buffer of data will be cropped from one channel to equalize recording lengths.

The first step in writing data to the board is to call the `start_write_cc()` function. This function takes no arguments, it merely configures the board and prepares it for a write operation. You may write data either from an open file pointer or from a memory buffer, with a similar interface in both cases. The `write_cc()` functions will return as soon as all data passed to them has been written to an internal buffer on the hardware. This does not necessarily mean that the operation has completed. After all data has been written to the board (through one or multiple `write_cc()` calls), execute the `stop_write_cc()` method. It will clean up the status of the board and wait for all data to actually be played to the user (real-time synchronization). The `write_cc()` functions may be used for either the master, slave, or both channels.

The `read_cc()` functions will abort on any dtmf key pressed, the `write_cc()` functions will only abort if the key pressed is in the user-specified abort string (a mask of dtmf characters). If such an abort occurs, the `get_last_dtmf_cc()` method may be used to determine which key caused the abort (it will return `NULL` if no abort was triggered). If, instead, you wish to simply wait for a dtmf key, use the `get_dtmf_cc()` function. It will wait for the specified amount of time (time-out is in seconds) for a dtmf key to be hit. It will only register those keys specified in the abort string (see the documentation for the `configure_cc()` method, Section 5.3). The function `generate_dtmf_cc()` is included in the interface for completeness, but is currently not implemented.

5.8. Example Diagnostic Programs

The example diagnostic programs described in Section 6.9 were all written using class Tele_interf. These programs exercise the basic board functionality.

The **simple/** subdirectory contains simple example programs demonstrating the system's basic functionality. There are four applications included with the system. The first, **wait_prompt_read**,

has a channel wait for a caller and record a short utterance of speech. The second, **dial_prompt_read**, is very similar to **wait_prompt_read**, the only difference being this system initiates the call to a specified telephone number. An example of dtmf (touch-tone keys) number input and output is available as **wait_prompt_pin_synth**. The final program, **wait_dial_cnf_read**, tests the conferencing capabilities of the board by connecting two users and recording both sides of a short conversation.

These four programs are simple examples of most basic telephony-board functionality. The programs have examples of call flow initiation, single and dual channel audio i/o, and number i/o. These basic components are most of what is used in the full parameter based data collection system. Complete understanding of these basic programs should be achieved before attempting to create more complex telephony applications in C++.

INSTALLATION 6

This section provides assistance on installing the full speech data collection system. It begins with a brief overview of the hardware installation. A few suggestions for the Linkon software installation are also provided. There are external software packages necessary to use the system most effectively. A detailed description is given to the building of the ISIP software onto your system. Lastly, testing programs are described that can assist in diagnosing specific hardware problems.

6.1. System Requirements

6.2. Hardware Installation

The Linkon and Newbridge hardware installation is covered in depth in the respective manuals, but we have a few suggestions.

Although a CSU is not required for the T1 line to interface with the Newbridge board, it is an invaluable diagnostic tool. In addition to diagnostics, the CSU assists in protecting the computer hardware from power surges along the T1 line.

NOTE: I don't know how to write this section

6.3. Linkon Software Installation

Although Linkon Corp.'s documentation describes installation of system software in detail, we wish to add our suggestions for fusing their code with your environment.

DDI vs. TVOX. Common interface to reboot board, how to set this to happen automatically (supposedly).

NOTE: I don't know how to write this section

6.4. Third Party Software Packages

While the software written is fairly independent, it does require a few packages to be installed (and visible) on your system. TCL, Tk, and Perl are necessary to run the application builder. We recommend having TCL/Tk version 8.0 or greater and Perl version 5.0. In the interest of making our software easier to install and maintain, no third party extensions to TCL/Tk are used, the standard distribution will suffice. TCL/Tk can be easily found on a variety of anonymous ftp sites, including *ftp://www.smli.com/pub/tcl*. The tk executable *wish* must be in your path for the application builder to run. The Perl distribution can also be found via anonymous ftp or the web, try *http://www.lafayette.edu/doughera/doughera/perl/perl.html*.

6.5. Installing ISIP code.

The first step in installing the software is to decide where to place the distribution on your file system. Place the file `linkon.tar` in the parent directory of where you wish the system to reside. For example, if you wish to have a directory /usr/local/isip/linkon, you would untar from /usr/local/isip, the linkon directory will be created automatically. The following commands would be necessary for this example:

```
mkdir /usr/local/isip;
mv linkon.tar /usr/local/isip;
```

```
cd /usr/local/isip;
```

Next, untar the distribution.

```
tar -xBf linkon.tar;
```

This will create the directory./linkon, placing all software within. In order to load the environment needed to either install or run the software:

```
cd linkon;
source ./LK_SETENV.sh
```

Next, build the software by executing the script

```
./Make.sh install
```

This will build all relevant software and install it as necessary. The software, as all of ISIP's public domain code, is written to be compiled with GNU gcc and GNU make; Make sure that GNU make is highest in your PATH. Also, the 'ar' command must be in your path. It can be found typically in /usr/ccs/bin/ar.

ISIP code is designed to be easy to install.

To make the software run correctly, it is necessary to set an environmental variable, LINKON, to point to the base directory of the installation. It is also recommended that the directory $LINKON/bin/$ISIP_BINARY be in your PATH. A modification to your ~/.bashrc will probably be necessary, adding the following commands.

```
LINKON=/usr/local/isip/linkon;
export LINKON;
source $LINKON/LK_SETENV.sh;
```

* assuming that /usr/local/isip is the base directory of your installation.

contact your UNIX systems administrator if you require assistance in this task.

6.6. Configuration of the T1 Line

set up RTANI on the channels. If you desire to run the system without this feature, you must edit the Tele_interf class header file (located in $LINKON/class/tele_interf/v1.0/tele_interf_constants.h). After making this change, you must rebuild the ISIP code.

Currently have a message on Kay Franklin's voice mail to get back to me

6.7. Echo on the Digital Network

Hybrid echo is the primary source of echo generated from the public-switched telephone network (PSTN). This is created as voice signals are transmitted across the network via the "hybrid" at the

2-wire/4-wore PSTN conversion points. This happens when a portion of the energy in a 4-wire section is reflected back on itself, creating the echoed speech. When the total network delay exceeds 36ms, echo is noticeable.

In recording conversations between two people, separate channels are used for each of the speakers. Ideally each channel contains signal from one of the speakers only. However, due to echo, each channel may in practice contain signals from both the speakers, though to varying degrees. Most audio recorded at ISIP over our T1 line has had no echo present, but this ends up being a product of the phone network, not the telephony hardware. The Linkon digital hardware has not introduced any cross-talk in our testing.

The process of echo cancellation involves two steps. First, as the call is set up, the echo canceler employs a digital adaptive filter which characterizes the echo path. When the signal passes through this filter, a replica of the echo is generated. If the filter characteristics match the echo path characteristics, a simple subtraction of the signal and the synthesized echo rids the signal of echo. The second process is that of error suppression, where the residual echo is attenuated to below the noise floor.

However, the cancellation is not perfect and an adaptive process is pursued where the residual signal is used to adapt the filter more accurately to the echo path. ISIP supports an off-line echo cancellation software that is very useful for two-channel data recorded from the digital telephone network[8].

6.8. Linkon's Analog System

Linkon Communications manufactures two versions of the fs4000 board.The board described and used in this document is the T1 board. They also manufacture a less expensive analog board, which directly interfaces to 8 analog telephone lines. The two systems are mostly identical from the software standpoint. In fact, much of the original code in this system was developed on the less expensive analog system. This was a cost effective option, offsetting the high usage cost of the T1 for basic system development.

Unfortunately, the software designed may not completely ignore the hardware platform. A few tasks are performed differently on the T1 line then the analog system. For one thing, lines on the digital system should be left in the offhook state when not in use, so as not to confuse the T1 hunt algorithm. Analog lines should generally be left on-hook. Obviously, no real-time ANI information will be available on the analog system, so modifications would be necessary to that code as well. Fortunately, the DSP chips on both boards handle data in much the same way, so most of the audio i/o functions could remain unchanged.

6.9. Basic Hardware Diagnostics

Several programs are available to help test the basic functionality of the hardware. All of these programs are located under the directory $LINKON/util/example_applications/v2.0/diagnostics/.

The simple/ subdirectory contains simple example programs demonstrating the system's basic

functionality. There are four applications included with the system. The first, **wait_prompt_read**, has a channel wait for a caller and record some speech. The second, **dial_prompt_read**, is very similar to wait_prompt_read except the system initiates the call to a specified telephone number. An example of dtmf (touch-tone keys) number input and output is available as **wait_prompt_pin_synth**. The final program, wait_dial_cnf_read, tests the conferencing capabilities of the board by connecting two users and recording both sides of the conversation. For more information on these programs, view Section 5.8.

If configuration problems persist with the hardware, it may be necessary to consult with your Linkon vendor. The Linkon software distribution includes several diagnostic programs. These programs are at a much lower level than those described in the previous paragraph, but are helpful to develop a common ground with a technical support representative. Before running a low level linkon diagnostic program, set the linkon software to report maximum debugging information,

```
lkonlog -b 0 -d fff
```

This command must be run as root. This level of debugging should not be necessary in most situations.

SWITCHBOARD 7

The first SWITCHBOARD database was collected at Texas Instruments in 1992 [1]. Dr. Godrey stated that large structured collections of speech and text are essential to progress in speech and speaker recognition research—an accepted truth in the speech community. At the time of this writing, the Linguistic Data Consortium (LDC) is in the process of collecting SWB-II phase 3.

To test the versatility our data collection system, we have implemented the LDC's SWITCHBOARD-II phase 3 protocol. The application parameter file was created using **lk_appbuilder**. Since this is the most complicated application designed on the system, this section of the document will go over a few key points of the interface. The modularly designed Oracle database access routines used in the SWB implementation also provide useful examples of how the system may be expanding to interface with other software.

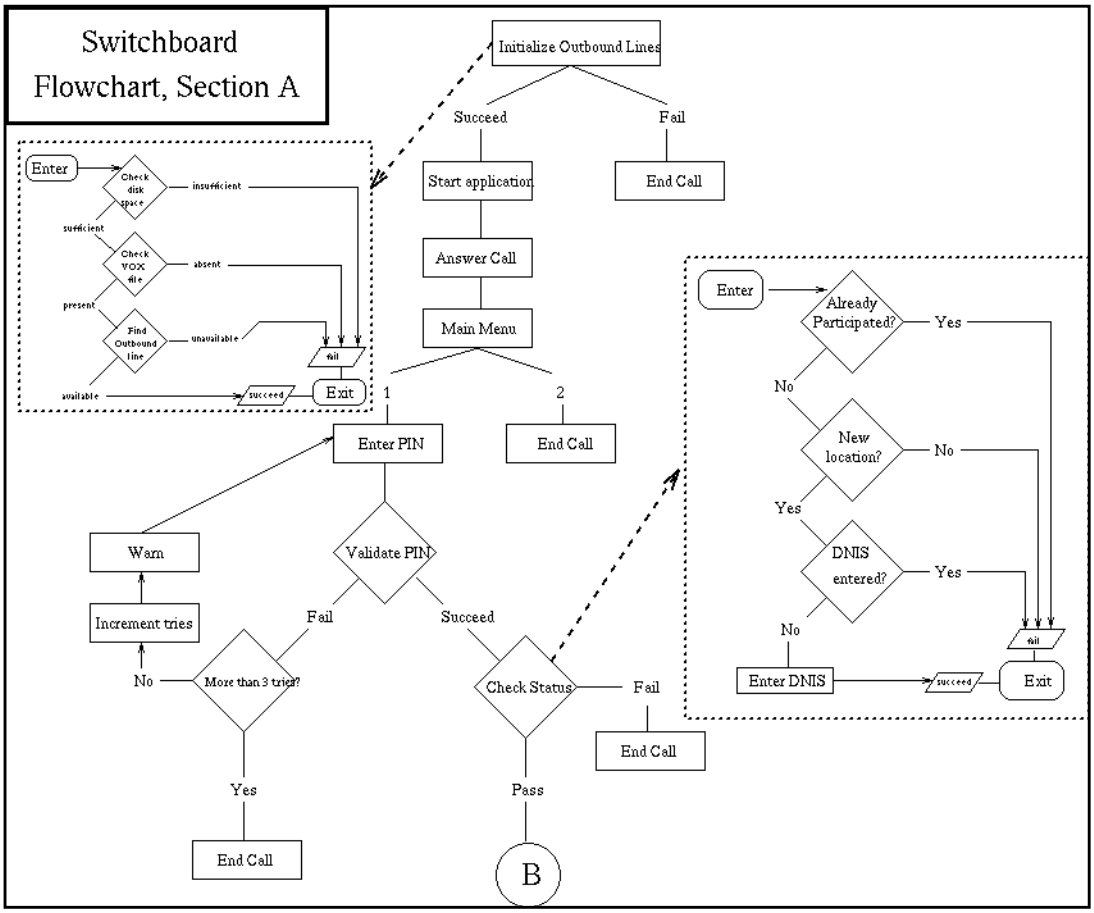


Figure 7.1: Flowchart for SWITCHBOARD (section A)

7.1. LDC's SWITCHBOARD Protocol

The LDC SWITCHBOARD protocol defines a very sophisticated data collection system. Many issues went into the design, many of them dated back to simpler data collection efforts years ago at Texas Instruments. Some of the queries are for legal issues (such as the 'press 1 to participate'), while other blocks assist in correcting physical shortcomings in the phone network (such as the problem ANI check).

To collect a SWITCHBOARD database, a large number of volunteers must register into a database, providing a list of availability times and phone numbers. The participants then call into the system at their leisure, using a pin number to identify themselves. A second caller will then be located by the system using the database of available participants. The system will then dial up to 5 phone numbers in an attempt to connect a second caller. Once both callers are on-line, a topic prompt will be read and the conversational speech will be recorded. The database is designed such that two participants will never talk to each other twice. Also, each call must be initiated from a unique phone number, as gathered through the RT-ANI packets from the digital phone network.

The Phase 3 of the SWB-II corpus was gathered from Southeastern universities during the fourth quarter of 1997. It was collected using the Intervoice system, described briefly in Section 2. Figures 7.1 and 7.2 describe the call flow of SWB-II phase 3.

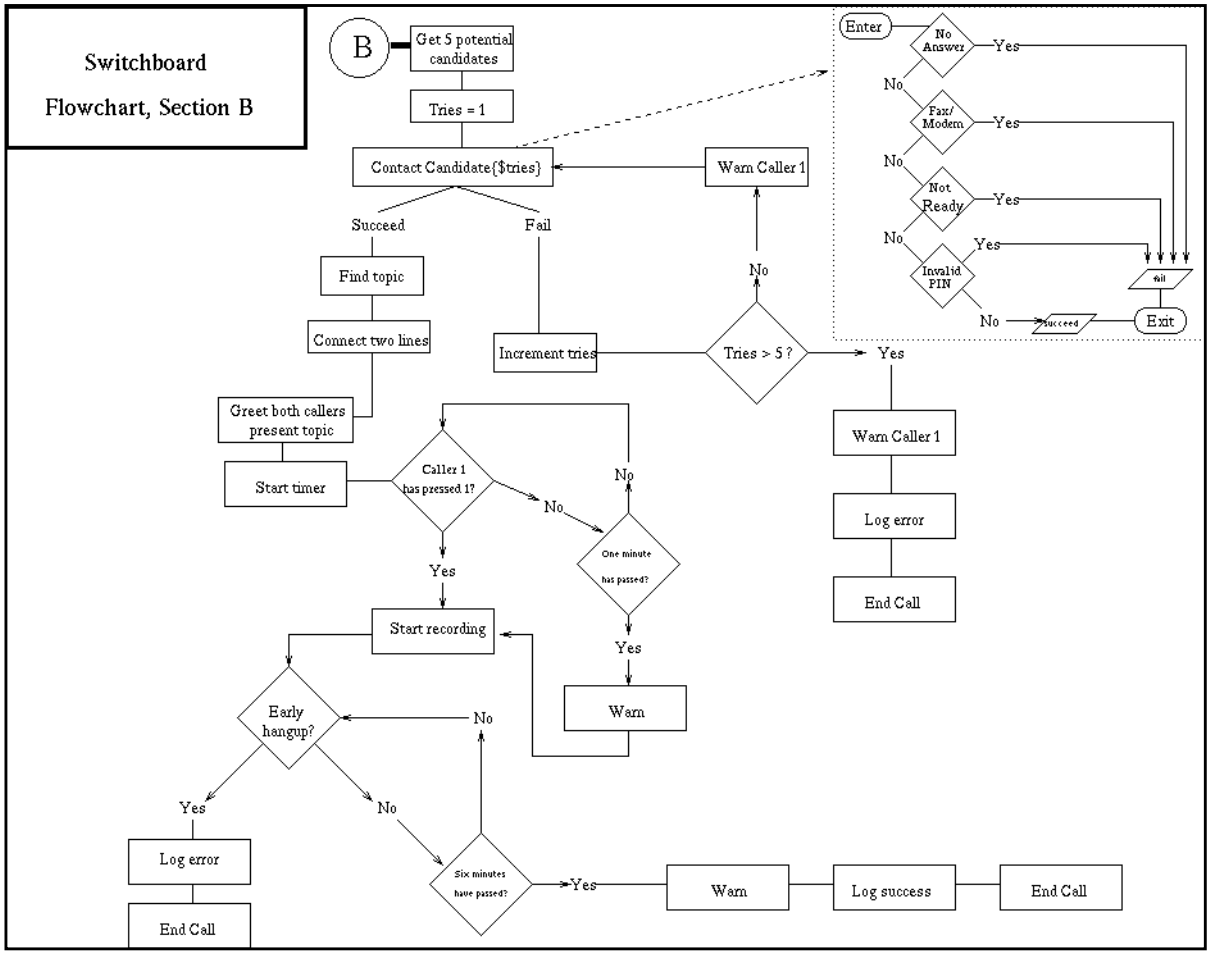


Figure 7.2: Flowchart for SWITCHBOARD (section B)

7.2. SWITCHBOARD Implementation

The LCD SWB-II phase 3 protocol has been implemented on our data collection system in order to test versatility in large applications. The application parameter file was created using **lk_appbuilder**. The full application, which almost perfectly mimics LDC's application running on the Intervoice hardware, is a 101 item application. All source code and data collected may be viewed in the $LINKON/pilot_swb/ directory, hereafter called $SWB_DCOL. The configuration file is in config/main.dcol. The graphical output of **lk_appbuilder** for the complete application is shown at the end of the chapter.

All database access is done remotely, by connecting to the same oracle server at the University of Pennsylvania that the Intervoice system queries. All such access are performed through perl scripts, interfacing to the data collection system through **shell command** items. Each of these perl scripts reads information from the data collection system to stdin, performs its task by connecting to LDC's oracle server, and writes back to the application through stdout. The perl scripts to perform specific database tasks are stored in $SWB_DCOL/config/scripts/ directory, underlying perl modules are in $SWB_DCOL/config/oracle/. This abstract interface description can be used to create a wide variety of database applications.

The database query interface is shown in Section 7.3. After the initial socket authentication, the first field returned by each query will contain a status value—either true(T), false(F), or not available(NA). If the oracle server is not available or network problems occur in reaching LDC, ‘NA’ will be returned. In this case, each program will terminate the current callers with an error message and alert the project manager through email. A return of ‘F’ implies that a logical error occurs (not admitting the caller, no free callers, no available topics, etc.), which should be handled by software. A ‘T’ return means that no problems occurred in this test. The call_stat query, executed after each speaker, requires an 11 value enumeration for <call_status>. The possible values for this field are listed in Figure 7.4. Not all of these return codes are used, more on this in the final discussion of this chapter.

The first shell command described in main.dcol starts a local database relay server. For efficiency, a single network socket is opened at the beginning of the call and remains open throughout the duration of the call. All database access is performed through opening local sockets to this relay server. Depending on network traffic between Mississippi State and the University of Pennsylvania and the current load on LDC’s oracle server, it takes up to five seconds to open the initial socket. To hide this latency from the user, the introductory prompt is played at this time. Directly after the first welcome message is played, a query is done using the caller’s ANI information. This is the only query that can be performed at this time, the secondary intention (outside of the protocol) is to test the database connection before asking the user to continue. If the server was not started properly, an error message will be played before the caller is asked for any input.

After the system is assured that the caller wishes to participate, the program branches if the caller’s ANI information returned from the phone network is among a manually maintained list of known problem numbers. If so, the caller is prompted to enter her phone number manually.

```
startup:
    () <= 130_18_6_2[1]
    (t/f/na) <= dbaction check_prob_ani <mani>
    (t/f/na) <= dbaction check_mpin <mpin>
    (t/f/na) <= dbaction check_mani <mpin> <mphone>
    (t/f/na) <= dbaction check_madmit <mpin>

get second caller:
    (t/f/na, <spin>, <sphone>) <= dbaction get_spinphone <mpin>

connect:
    (t/f/na, <topic_id>[2]) <= dbaction get_topic <mpin> <spin>

callstat: executed every time
    (t/f/na) <= dbaction call_stat <date> <mpin> <mani> <mphone> <spin> <sphone>
              <topic_id> <off_hook_duration>[3] <talk_duration (connect-disconnect)>
              <record_duration> <call_status> <channel_num>
```

1.I.P. address of isip02, our local server running the data collection system. This number served as a simple authentication double check.
2.Use topic id number to create prompt filename.
3.All times in minutes:seconds (MM:SS).

Figure 7.3: Remote Oracle query specification

The next step in the application is to test the identity of the caller through a personal identification number (PIN). The caller is given up to 3 chances to enter a valid PIN. Once the PIN has been tested, the system will check the uniqueness of the caller's telephone number—since the database protocol mandates that each caller may only participate once on the same phone line. The final test before looking up a second caller verifies that the caller has not participated within the last 24 hours. These queries are performed remotely to the LDC Oracle server.

code	value
0	CallSucceeded
1	Busy
2	CalleeNotAvailable
3	RingNoAnswer
4	AnsweringMachine
5	FaxOrModem
6	CallIncomplete
7	CalleeHangUp
8	CallerHangup
9	TimeOut
10	LineProblem

Figure 7.4: Call_status field enumeration

Now that the first caller has been processed, a second caller must be located from the list of available participants. A database query returns a pin and phone number set. It dials the second caller, asks for the participant by name, and asks for a pin. The second user (called the slave) is given up to 3 chances to enter a valid pin, which is compared to the value returned by the original query. If the second participant is not available or does not enter a valid pin, another caller will be attempted. Up to 5 callers will be tested in this manner. If none are available, an error message will be sent to the first caller. This last query will also end the local database server process.

Once both callers are on-line and verified, a welcome message and brief instructions are played to both parties. After a topic prompt is played, the two callers are connected. After one minute of introductions, both sides of the conversation are recorded. After this recording, both callers are thanked and the call is ended. A final call_staus query is given to the oracle server to end the call.

There are only two inconsistencies between the Intervoice implementation and this implementation. The first difference is a more limited call status code—our system only outputs three values (connected, no answer, or busy), while the Intervoice system outputs 11 values (indicating specific trunk and line problems). This information is available from the linkon hardware, but changes to the dial item would be necessary to provide this level of information. The second problem is reading dtmf characters during the playback of an audible prompt. This method is simply not fast enough on the Linkon hardware, so it is not used in the full interface.

A feasible implementation of the LDC SWB-II phase 3 protocol has been developed on our UNIX based telephone data collection system. The database aspects of the application will prove very helpful for other applications, as the perl scripts are written in a modular fashion and are highly reusable. All levels of the system, from the hardware to the application builder, have proven robust through strenuous local testing. The intense system design also provided valuable feedback to the design process—the SWB exercise has vastly improved the data collection system.

8. CONCLUSIONS

We have developed a robust, fully-expandable system for platform-independent collection of telephone speech data. Our object-oriented software libraries and easy-to-use GUI provide powerful tools with which even a novice user can efficiently create complex applications. This document provides a detailed overview of this system, as well as a tutorial to illustrate prototyping of applications. While the current focus of the speech research community is recognition of broadcast news (HUB-4), significant effort is still expended towards SWITCHBOARD-type telephone data, for which this system will play a pivotal role.

A software system of this magnitude can, of course, be implemented in numerous ways. The code-base has undergone multiple revisions and additions as the program definition has evolved. Also, extensive testing with the SWITCHBOARD application brought many practical bottlenecks to the surface. As the main phase of the Linkon project draws to a close, several ideas for expansion of the current system come to mind. For instance, adding an include directive to the parameter file would provide several benefits, including basic modularity and an easy mechanism to re-use pieces of code and faster editing of the resulting smaller parameter files. A more powerful interface to the application builder's debug mode would be a helpful addition. While the SWITCHBOARD application we developed in-house proves the feasibility of the system for a serious application, these additions would only serve to strengthen the usability of the system.

In addition to the software expansions mentioned above, it would be interesting to migrate the system to a new set of hardware. Specifically, an ATM interface to the phone network may prove to be more cost effective for high call volumes in the years to come. The code was specifically written with such a transition in mind, but undoubtedly issues will arise in such a transition. The only true measure of hardware independence is the effort involved in porting a system.

Adherent to ISIP's commitment to public domain software, this document and the complete software system is available at *http://www.isip.msstate.edu/resources/technology/projects/1997/t1_interface.*

9. ACKNOWLEDGEMENTS

We wish to thank Jack Godfrey[1], Mark Liberman, Dave Graff, Kevin Walker, George Zipperlen, and all of the staff at the Linguistic Data Consortium for funding and supporting this work as well as for their significant technical assistance throughout the course of the project. Vicki Smith, Kay Franklin, and the network support team at MCI were instrumental in helping us decipher our digital interface to the phone network. Anna M. Linley, George Dinsdale, and Jack Lin of Linkon Corporation should also be recognized for hardware technical support at the beginning of the project.

The first student engineer to work on this project was Paul Kornman, one of the original members of ISIP. He joined ISIP in Fall'94 as a Junior, one of the finest undergraduates in the ECE department. It was partly Paul's zeal for telecommunication networks (he was also trying to

1.Dr. Godfrey has moved back to Texas Instruments during the course of the project.

connect his apartment complex with a T1 line) that inspired us to take on such a project. Paul Kornman was killed in an auto accident by a drunk driver in the early morning on Tuesday, October 17, 1995. True to Paul, he was driving home at 1 AM after studying late into the night at his office in Simrall Hall. Paul was an excellent engineer and a good friend who will be missed.

10. REFERENCES

[1] J.J. Godfrey, E.C. Holliman, and J. McDaniel, "SWITCHBOARD: Telephone Speech Corpus for Research and Development," in *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. I-517-I-520, San Francisco, California, USA, March 1992.

[2] B. Wheatley and J. Picone, "Voice Across America: Toward Robust Speaker Independent Speech Recognition For Telecommunications Applications," *Digital Signal Processing: A Review Journal*, vol. 1, no. 2, pp. 45-64, April 1991.

[3] J. Picone, "Managing Software Complexity in Signal Processing Research," in *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. III-41-III-44, Minneapolis, Minnesota, USA, April 1993.

[4] J.E. Porter, "Features of T1 Line CODEC Code Distributions," ITT Aerospace/Communications Division, 10060 Carroll Canyon Road, San Diego, CA 92131, July 1991.

[5] "Java Speech API: A White Paper," available at http://java.sun.com/products/java-media/speech/, Sun Microsystems, 1998.

[6] Linkon Communications Board Direct Driver Interface Programmer's Guide and Reference Manual. Release 5.0.0, July 17, 1996. Linkon Corporation.

[7] J. Fiscus and J. Garafolo, Speech Header Resources (SPHERE), http://www.itl.nist.gov/div894/894.01/software.htm

[8] A. Ganapathiraju and J. Picone, "Echo Cancellation For Evaluating Speaker Identification Technology." Proceedings of IEEE Southeastcon, pp. 100-102, Blacksburg, Virginia, USA, April 1997.

APPENDIX A. ITEM DEFINITIONS

A data collection system parameter file is made up of two parts, general parameters and items. The general parameters, described in Appendix B, are parameters common to and possibly used by each item in the system. The actual application, however, is designed by linking a list of items together. This section describes the behavior of items, the basic building blocks of database applications.

This is an object oriented system, so user data is always kept WITHIN the individual items themselves. The only two types of data that can be passed between objects are character strings and filenames. To access dynamic data that has been set earlier in the program, simply refer to the name of the item in which the data was collected. Each type of item interprets the data differently, as evident in the different “input data” information.

The normal control flow of a program is to traverse each item in order. This leads to very static systems, so the system needs additional parameters in order to be capable of more complex flows. The flow altering mechanism provided by the system can be simplified to *if ERROR jump NAME*, where name is the name you assign to another item. This basic structure can be built upon to provide very complex programs.

These are the available item types:

Item: wait for call

Description:

This item puts the line in an idle (on-hook) state until an in-bound call arrives. Upon arrival of the call, the line comes off-hook and the call is answered. The RT-ANI information (caller-id) of the user who called is stored and can be referenced whenever required.

Parameters:

none.

Output:

Dtmf is set to the phone number that dialed in.

Conferencing:

Can be performed on either the master or slave channel.

Item: dial

Description:

This item performs a hook flash and then dials the specified number. When someone on the other end of the line speaks a word, the item is considered to be successful and it moves on. If a timeout occurs (no answer), the application aborts and tries to jump to the specified operation.

Parameters:

number to dial (referenced) (*ref_name*):

The telephone number is referenced from this item. If this is specified, the statically expressed telephone number specified in the dtmf parameter is ignored and instead, the system dials the dtmf string stored in the item named here. This should typically be set to reference an item of type get dtmf or a shell command. The data flow can be seen visually as the bottom arrows of the plot, data is “pulled” from its origin to where it is used.

number to dial (static) (*dtmf*):

This is the telephone number that is dialed. This can be left NULL and the reference object can be set to dial the numbers set in another item, such as a “get dtmf” or a “shell command” item.

connecting message (*filename_0*):

This parameter can be to a NIST SPHERE file containing an audio message such as “Connecting now.” It can be played in the future if appropriate by using a play item configured to reference the dial item.

busy message (*filename_1*):

This parameter can be set to a NIST SPHERE file containing an audio message such as “The number was busy, please try again later.” It can be played in the future if appropriate by using a play item configured to reference the dial item.

no connection message (*filename_2*):

This parameter can be set to a file containing an audio message such as “No connection was made.” It can be played in the future if appropriate by using a play item configured to reference the dial item.

jump to if no connection (*jump_name*):

This parameter controls the behavior of the item on an unsuccessful dial. If the dial attempt fails (no connection is made), control aborts to the item with this name. If no value is specified here, control advances to the next item regardless of exit status. The control flow can be seen visually by the arrow leaving the top of the dial item.

Output:

out_filename is set to the appropriate status file. A good use of the output from a dial object would be to create a new play item after the dial object. By configuring the play item with the dial item as its referenced item, the audible dial status will be played.

Conferencing:

Can be performed on either master or slave.

Item: hangup

Description:

This item terminates the current connection by hanging up the line. After the hook flash, the line is left in an idle (off-hook) state.

Parameters:

goodbye message (*filename_0*):
This parameter can be set to a file that is played to the user before hanging up the line. It is played, of course, only if the user is still on the line and has not hung up.

Output: none.

Conferencing:

Can be performed on either the master or slave channels.

Item: play

Description:

This item plays the specified audio file to the line. The audio file should be in the NIST SPHERE file format, set to 8kHz, 8 bit u-law data. The signal is normalized before playing through scaling by the sample_max SPHERE header entry.

If the max number of digits parameter is set to a non-zero value, then playback is aborted on any key-press (effectively ignoring this parameter). The abort keys parameter will instead be used to specify the TERMINAL character for a string of digits. If the max number of digits parameter is greater than 1, the hardware waits for the specified amount of time after a key-press aborts playback for further dtmf keys.

Parameters:

filename to play (static) (*filename_0*):
This parameter specifies the NIST SPHERE filename to be played.

play abort keys (*abort_keys*):
These are various dtmf keys that make up the dtmf interrupt mask. Any of these keys being hit causes the current item to abort playback. If a key is not specified in this list, the board does not respond to it as a user input. If the max number of digits parameter is set to a non-zero value, then playback is aborted on any key-press (effectively ignoring this parameter). Instead, the abort keys parameter is used to specify the TERMINAL character for a string of digits of digits (if n > 1).

max number of digits (*speaker_mode*):
This is the maximum number of digits to be read from the dtmf (touch tone) keypad. This is a useful parameter if the number of digits to be read from the user is known in advance. For a menu situation, setting this to ‘1’ causes only one digit to be read and then the system moves on instantly, waiting no longer for the next character.

wait time between digits (*wait_interval*):
This is the maximum amount of time that this item waits for between dtmf keys. If an event does not come in this interval, the item aborts and waits no longer for further digits.

filename to play (referenced) (*ref_name*):
Instead of specifying a static filename to be played, the user can reference the filename of another item. The referenced item should usually be of type record (i.e. play the newly recorded audio file) or shell command (i.e. play the file specified by the FILENAME: header in the program output). The data flow can be seen visually as the bottom arrows of the plot, the data is “pulled” from its origin to where it is used.

jump if dtmf keys are hit (*jump_name*):

This item jumps to the named item if the following conditions are met: 1. this parameter points to a valid name 2. the abort keys parameter is not null 3. the number of digits is greater than 0 4. a dtmf key is hit during playback

Output:

The dtmf string read is available as output data.

Conferencing:

Can be performed on either the master, slave, or both channels.

Item: record

Description:

This item records audio data over the line into a NIST SPHERE file. There are two ways to specify when recording will stop:

1. use the signal detector --- this will cause data to be recorded as long as a signal is present (within specified parameters, see Appendix D).
2. use the wait time to specify the duration of recording, thus disabling the signal detector.

Parameters:

filename template (*filename_template*):

The characters in this field are added to the internal program data (including the speaker number) to create a unique filename. Note that the user may use the same filename template only once within an application.

speaker mode (*speaker_mode*):

This field is copied directly into the recorded NIST SPHERE file. Commonly used values for this field are “read,” “prompted,” or “conversational.”

Output:

the main recorded file is saved in out_filename, the length of recording (in seconds) is stored into dtmf.

Conferencing:

Can be performed on either master, slave, or both channels. In the conference record mode (*both*), two files will be created. The filenames will be calculated by combining the filename_template and the channel number. It should also be noted that a conference record will always return equal duration recordings in the two channels, even if cropping one channel is necessary.

Item: synthesize

Description:

This item causes the data string of numbers to be converted into an audio signal and played over the line. Non-numeric characters in the data string are ignored.

Parameters:

abort keys (*abort_keys*):

These are various dtmf keys that make up the dtmf interrupt mask. Any of these keys being hit causes the current item to abort its action. The hardware does not abort an item for a key not specified in this list.

filename for digit 0 (*filename_0*):

This is the filename (including full pathname) for the ‘0’ digit NIST SPHERE file. From this filename all other digit filenames are permuted by substituting the last ‘0’ with ‘1’ through ‘9’, and optionally ‘11’ for a pause (see the speaker_mode parameter for details).

telephone number? (*speaker_mode*):

This parameter is used to tell the synthesize item to play the number as a telephone number (e.g. 1-601-555-1212 vs. 16015551212). Placing a true value (non-null) in this space causes the appropriate pauses to be inserted in the playback (the default behavior is to play a number with no breaks in the output).

data string (static) (*dtmf*):

This field is used to statically set a string. The characters in this field commonly correspond to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can also be specified.

data string (referenced) (*ref_name*):

This field is used to reference a string from another item. If the user specifies the name of an item that has a dtmf parameter set (e.g. get dtmf, shell command, play, wait for call, iterate, etc.), then the value in this item’s string container (dtmf) is used. This string of characters commonly corresponds to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can also be specified. The data flow can be seen visually as the bottom arrows of the plot, data is “pulled” from its origin to where it is being used.

Output: none.

Conferencing:

Can be performed on either master or slave.

Item: get dtmf

Description:

This item reads a string of dtmf (touch-tone) characters entered by the user and stores them to internal data. These characters can be referenced for use by other items, see help on referencing data for more information.

An end of string event terminates the reading of data. The end of string can be generated in three ways:

1. a timeout waiting for a dtmf key (a key-press before timeout occurs resets the wait)
2. number of characters read exceeds the value of the max number of digits parameter
3. a terminal character can be specified through the abort_keys parameter

The end-of-string terminal character is not returned as part of the string.

Parameters:

terminal key (*abort_keys*):

The get dtmf item reads dtmf keys from the keypad, and concatenates all the accepted key-presses to the current string. This field is used to specify the TERMINAL character in such a string of dtmf key-presses.

key timeout (*wait_interval*):

This is the maximum amount of time that this item waits for a dtmf key to be hit. If an event does not come in this amount of time, the item aborts its action. If it has jump capabilities, it performs the jump in this situation. Note that the board waits for this length of time between each digit. It only tries to alter the control flow if no keys are hit. The control flow can be seen visually by the arrow leaving the top of the item's icon.

max number of digits (*speaker_mode*):

This is the maximum number of digits to be read from the dtmf (touch tone) keypad. This is a useful parameter if the number of digits to be read from the user is known in advance. For a menu situation, setting this to '1' causes only one digit to be read and then the system moves on instantly, waiting no longer for the next character.

short circuit data (referenced) (*ref_name*):

This field is used to reference a string from another item. If the user specifies the name of an item that has a dtmf parameter set (e.g. get dtmf, shell command, play, wait for call, iterate, etc.), then the value in this item's string container (dtmf) is copied into the data container for this get dtmf item as if it was entered on the keypad. Control is advanced immediately to the next item. This feature is used to create the common task block of

prompting for and reading a string from the user's dtmf keypad. The play item can be configured to read dtmf keys if they are hit during playback, but it does not wait after the end of the file if a key has not been hit. The get dtmf item is configured immediately following the play prompt to try to import the string read. If the user begins to input a string during the prompt, the entire string entered is copied into the data string container of this get dtmf item. Later in the application, the user need only to reference the data string held by this get dtmf item to read the data entered from either the play prompt or this get dtmf item. This string of characters commonly corresponds to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can be specified. The data flow can be seen visually as the bottom arrows of the plot, data is "pulled" from its origin to where it is being used.

jump to on time out (*jump_name*):

If no valid dtmf key is hit within the time limit, control aborts to an item with this name. The control flow can be seen visually by the arrow leaving the top of the item's icon.

Output:

The string of characters is saved into dtmf

Conferencing:

Can be performed on master, slave, or both. If the both option is used, only one string is kept, both channels being merged in the order in which the keys were hit.

Item: check dtmf

Description:

This item makes a comparison test of two character strings. If the match fails, the control flow jumps to a specified item. If the match is successful, the control flow passes to the next item as expected.

This can be visualized by plotting the state graph.

If you wish to implement a "jump to if match" system, have a "direct jump" item immediately follow this item, and have it jump to the match item. Then configure the current item to jump to the successor of the jump item.

Parameters:

list of numbers (*filename_0*):

This is a file containing a list of strings that is compared against the referenced dtmf string. If the referenced data string does not match any of the strings in this file, the comparison fails and the item jumps to the specified item. A successful match results in normal program flow to the next item.

specific number (*dtmf*):

This is a specific data string that can be compared against the referenced dtmf number string. If the referenced data string does not match any the specified string, the comparison fails and the item jumps to the specified item. A successful match results in normal program flow to the next item. The control flow can be seen visually by the arrow leaving the top of the check dtmf item.

string to check (referenced) (*ref_name*):

This field contains the name of the item who’s data is of interest in the comparison function. It is most likely an item of type get dtmf, shell command, or iterate. The data flow can be seen visually as the bottom arrows of the plot, the data is “pulled” from its origin to where it is used.

jump to if no match (*jump_name*):

If the comparison of the referenced data string and the string(s) specified in this item fails, control aborts to the item specified with this name. The control flow can be seen visually by the arrow leaving the top of the check dtmf item.

Output: none.

Conferencing:

N/A.

Item: save number

Description:

This item saves a data string to a disk file. The filename is generated in the same way as that for the record item, see help on filename_template for more information. This is a very useful mechanism for databases to associate sequentially assigned speaker numbers with preassigned PINs or speaker identification numbers.

Parameters:

data string (referenced) (*ref_name*):

This field is used to reference a string from another item. If the user specifies the name of an item that has a dtmf parameter set (e.g. get dtmf, shell command, play, wait for call, iterate, etc.), then the value in this item's string container (dtmf) is used. This string of characters commonly corresponds to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can also be specified. The data flow can be seen visually as the bottom arrows of the plot, data is "pulled" from its origin to where it is being used.

filename template (*filename_template*):

The characters in this field are added to the internal program data (including the speaker number) to create a unique filename. Note that the user may use the same filename template only once within an application.

Output:

none.

Conferencing:

N/A

Item: set number

Description:

This item alters the internal dtmf string data of another item to the specified value. It is most commonly used to specify the name of an inner loop counter to reset counts during execution of outer loops.

Instead of the typical items that only read data from another item, this item writes data to another item.

Some caution is advised in using this item. Some items keep permanent data and will not be reset across calls, and therefore should not be processed with this item. For example, changing the compare value in a check dtmf item could cause a valid number to be rejected.

If the value specified is “reset iteration count” and the referenced item (or target item) is of type iteration, then the iteration will reset to its initial value.

Parameters:

target item (referenced) (*ref_name*):

This field contains the name of the item in which data is set. It is most commonly used to specify the name of an inner loop iteration counter, to reset it outside of the nested loop. The data flow can be seen visually as the bottom arrows of the plot, data generally appears to be “pulled” from its origin to where it is used. In case of the set number item, however, data flows in the opposite direction of the arrow.

value (*dtmf*):

This filed contains the value to be exported to the named item. The dtmf parameter of the named item is then altered to reflect the value specified here. Use of this item is the only way to externally affect another item’s data. This item can also be used to reset an iteration item to its initial value. This can be done by either explicitly specifying the initial value in this field; or by using the key phrase “reset iteration count”, in which case the iteration item will reset itself to its initial value. The latter method is preferred, keeping the reset value with the iteration item rather than this reset item.

Output:

The dtmf parameter may be accessed in the normal manner.

Conferencing:

N/A

Item: conference connect

Description:

This item creates a full duplex teleconferencing connection between the master and the slave channel. As long as a conferencing connection is made, no audio output operation (e.g. play, synthesize etc.) may be performed. Record, get dtmf, and all call flow items can be executed as normal.

Execution of this item has no effect if a teleconferencing connection already exists.

Parameters:

none

Output:

none

Conferencing:

Always performed on master channel.

Item: conference disconnect

Description:

This items causes a teleconferencing connection to be broken. This will allow audio signals to once again be output to the channels.

Execution of this item has no effect if no teleconferencing connection currently exists.

Parameters:

none

Output:

none

Conferencing:

Always performed on master channel.

Item: shell

Description:

Execute the command specified by the shell_command parameter. The program is run with no command line arguments, so it is often necessary to write a shell script interface to use common utilities. The input to the program is:

"<speaker_number>|<channel>|<dtmf>"

Where speaker_number is the sequential speaker number of the current caller, channel is the line number associated with the master channel, and dtmf is the referenced data string.

If the command is run in the foreground, program output is saved in a way that facilitates easy referencing by other items, such as check dtmf.

This item aborts (and tries to jump) if the external program returns ISIP_ERROR only in interactive mode.

Parameters:

command to execute (*shell_command*):
This is an external shell command that is executed by the system.

background process? (*speaker_mode*):
This parameter is used to specify whether the shell command should be run in the background or interactively. If the command is run in the background (non NULL value), control returns immediately and advances to the next item without waiting for any output or exit status. If this field is left NULL, the command is run interactively. Such an interactive shell command waits until the program completes execution, stores the program output, and possibly jumps if the program exits with an error status.

data string (static) (*dtmf*):
This field is used to statically set a string. The characters in this field commonly correspond to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can also be specified.

data string (referenced) (*ref_name*):
This field is used to reference a string from another item. If the user specifies the name of an item that has a dtmf parameter set (e.g. get dtmf, shell command, play, wait for call, iterate, etc.), then the value in this item's string container (dtmf) is used. This string of characters commonly corresponds to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can also be specified. The data flow can be seen

visually as the bottom arrows of the plot, data is “pulled” from its origin to where it is being used.

jump to on error (*jump_name*):

If the shell program encounters an error (exit status of ISIP_ERROR), control aborts to an item with this name. The program can more reliably return an error status by printing the string “RETURN: $ISIP_ERROR” to stdout. The control flow can be seen visually by the arrow leaving the top of the shell item.

Output:

dtmf holds the program output.

Conferencing:

N/A.

Item: mail

Description:

Send mail about an event. The /bin/mail standard Unix sendmail program will be used. Mail will be sent to the admin alias mail address specified in the general parameters. The following headers will be included in the message:

SPEAKER_NUMBER: <sequentially assigned speaker number>

DTMF: <string of characters to include>

REF-DTMF: <string of characters from referenced item (if any)>

The contents of “file to include” and the referenced output file will be will be mailed, each wrapped in:

“\n---------- begin File %s ----------\n”

“\n---------- end File %s ----------\n”

where %s is the file name.

Parameters:

email address (*address*):

This is the email address that the email generated by this item will be sent to. This address can either be set to a single-user address, an alias or to a space-separated set of email addresses. e.g. single user: address = admin@mydomain.edu, list: address = admin1@domain1.edu admin2@domain2.com

file to include (*filename_0*):

The contents of the file specified by this parameter, and those of the referenced output file are mailed, each wrapped in: “\n---------- begin File %s ----------\n”, “\n---------- end File %s ----------\n”, where %s is the file name.

data string (static) (*dtmf*):

This field is used to statically set a string. The characters in this field commonly correspond to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can also be specified.

data string and/or file (referenced) (*ref_name*):

This field is used to reference data from another item. If the user specifies the name of an item that has a dtmf parameter set (e.g. get dtmf, shell command, play, wait for call, iterate, etc.), then the value in this items string container (dtmf) is used. This string of characters commonly correspond to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can be specified. If the named item has the filename set, the contents of this file are included in the mail message. The data flow can

be seen visually as the bottom arrows of the plot, data is “pulled” from its origin to where it is used.

Output:

none.

Conferencing:

N/A.

Item: direct jump

Description:

Unconditionally jump to the named item.

This is useful in applying modularity to applications.

Parameters:

jump to item (*jump_name*):
Control is passed to an item named in this field. The control flow can be seen visually by the arrow leaving the top of the item's icon.

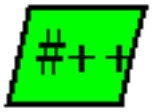
Output:

none.

Conferencing:

N/A

Item: iterate



iterate

Description:

This item holds an internal count value. Each time this item is executed (including the first time), the count value is incremented by the step value. Both step and the internal count must be numerical values.

This item is meant to be used in conjunction with check dtmf so as to limit the number of times a user can attempt to enter a valid pin number, to prevent security risks.

Parameters:

initial value (*initial*):
This is the initial value for an iterate item. A set number item can reset an iterate item to its initial value by using the key phrase “reset iteration count” instead of specifying a numeric value.

step value (*step*):
This is the step value for an iterate item. This is added to the internal count held by the iterate item (in the dtmf data string container) every time the iterate item is executed, including the first.

Output:

The present count is available.

Conferencing:

N/A

Item: kill program

Description:

This item kills the application. After execution, it will accept no more calls, the UNIX process will end, and the allocated channels will be freed to be used by other applications. This is meant to be available as an administrative item to allow a clean exit method. This will probably not be used in the final version of applications, but is a very helpful tool for development.

Parameters:

none.

Output:

none.

Conferencing:

N/A

APPENDIX B. GENERAL PARAMETERS

A data collection system parameter file is made up of two components, general parameters and items. The general parameters, described in detail in this section, are parameters common to and possibly used by each item in the system.

number of items num_items

This is a very generous estimate for the total number of items that may be in the system. This number MUST be greater than or equal to the actual number of items in the system. This field defines both the memory allocated at the start of the system and the length of each log entry. Only applications that have the num_items set to the same number can run on a common set of log files. If a small tolerance is allowed in this parameter, minor changes to the parameter file do not render previous log files unusable.

database name database_name

This field forms a part of the header of all SPHERE speech data files generated by the data collection system using the SPHERE field “database_name.”

monitor command monitor_command

The monitor program is useful for a system meant to run continuously (most data collection systems). This program will be run automatically upon the system being started and will monitor the current system processes, making sure that your application stays among them. If the program is to die, mail will be sent to the email address specified by the admin_alias parameter. An additional parameter used by this system is the program_name. This text string allows the monitor to distinguish between multiple processes running on the same system. Both the monitor command, the admin alias, and the program name must be set in order for the monitor to run. Leaving the monitor command parameter blank causes no monitor to be run.

program name program_name

This is the name that will be associated with the run-time process, and the name used to identify the program in the admin mail sent, if the program were to die. This need not be set if the monitor is you do not expect to use the monitor.

Administrative mail alias admin_alias

This is the email address (or address list) to which email will be sent during serious program diagnostics. Program termination (as reported by the optional run_monitor) or the data disk filling up are the only two events currently supported.

speaker directory format speaker_directory_format

This is the format that will be used to create the data directories for the speakers. It is used as: sprintf((char*)data_dir,speaker_directory_format,(int)speaker_number); Note that it uses the ANSI C sprintf function, formatted to include one long integer as data. See the C documentation on sprintf for more information. It is advisable to force leading zeros onto this format. For a five digit number, this is accomplished by using “%5.5d” instead of “%5d”.

needed disk space disk_info

This is the amount of free disk space (in megabytes) that is needed per call. If this amount is not available, the system will not accept any new callers and try to inform by email.

data collection system data_collection_system

This is the main data collection system variable. For the linkon system, this corresponds to the channel number. Leaving this value blank will allow the system to use the first available channel for the application. If the correct channel number is not available from the choices, please consult the User’s guide about configuring the channels in the GUI. For a two channel application it is necessary to specify both channel numbers.

teleconferencing system data_collection_system

This is the main data collection system variable. For the linkon system, this corresponds to the channel number. Leaving this value blank will allow the system to use the first available channel for the application. If the correct channel number is not available from the choices, please consult the User’s guide about configuring the channels in the GUI. For a two channel application it is necessary to specify both channel numbers.

database logfile database_logfile

This is the log file for the database being collected. It is a binary file that can be read but should not be edited by the user. Since the size of each entry corresponds to the num_items parameter, changing this parameter requires a new log file to be used. It is probably not a good idea to keep this log file on the same disk as the incoming data. It is possible to collect the same database over multiple processes (and phone lines). The system parses to the end of the log file and locks each new speaker number as it needs them. The lockf() protocol is used, which is secure even across NFS file systems. In order to run multiple instances of the same application, only the data collection system parameters need to be changed. If this file does not exist on program start-up, the application creates it and makes the first entry as specified by the starting speaker number parameter.

starting speaker number starting_speaker_number

Upon start-up, it is often preferable to start with a speaker number other than zero. The system will automatically start at the next available number from the log file if it exists, but it is sometimes desirable to use a new log file, or to cause a break (i.e.: starting at 1000 after a crash in which the last data collected was 843). If this parameter is not set or set to -1, it uses the next available slot. If this parameter is set to a number that has already been collected in the data file, the program will error (thereby preventing data overwrite).

progress file progress_file

Through the use of the file specified by this parameter the application builder's progress may be interfaced to a currently running instance of the same application. Viewing the application builder in this mode is the easiest way to debug a call flow, as the system will automatically set the current item to be the item currently being executed in the system.

APPENDIX C. ITEM PARAMETERS

abort keys abort_keys

These are various dtmf keys that make up the dtmf interrupt mask. Any of these keys being hit causes the current item to abort its action. The hardware does not abort an item for a key not specified in this list.

background process? speaker_mode

This parameter is used to specify whether the shell command should be run in the background or interactively. If the command is run in the background (non NULL value), control returns immediately and advances to the next item without waiting for any output or exit status. If this field is left NULL, the command is run interactively. Such an interactive shell command waits until the program completes execution, stores the program output, and possibly jumps if the program exits with an error status.

busy message filename_1

This parameter can be set to a NIST SPHERE file containing an audio message such as "The number was busy, please try again later." It can be played in the future if appropriate by using a play item configured to reference the dial item.

command to execute shell_command

This is an external shell command that is executed by the system.

conference conf_channel

The value of this field determines whether the conferencing mode is used or not. If this is set to *master* or "", then it uses the master channel only. If set to *slave*, then only the slave channel is used for this operation. If set to *both*, then both channels are used simultaneously for the operation. In the plot, M = *master*, S = *slave*, and B = *both*.

connecting message filename_0

This parameter can be to a NIST SPHERE file containing an audio message such as "Connecting now." It can be played in the future if appropriate by using a play item configured to reference the dial item.

data string (referenced) ref_name

This field is used to reference a string from another item. If the user specifies the name of an item that has a dtmf parameter set (e.g. get dtmf, shell command, play, wait for call, iterate, etc.), then the value in this item's string container (dtmf) is used. This string of characters commonly corresponds to dtmf keys on a touch-tone keypad, but any ASCII

character (such as program arguments) can also be specified. The data flow can be seen visually as the bottom arrows of the plot, data is “pulled” from its origin to where it is being used.

data string (static) dtmf

This field is used to statically set a string. The characters in this field commonly correspond to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can also be specified.

data string and/or file (referenced) ref_name

This field is used to reference data from another item. If the user specifies the name of an item that has a dtmf parameter set (e.g. get dtmf, shell command, play, wait for call, iterate, etc.), then the value in this items string container (dtmf) is used. This string of characters commonly correspond to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can be specified. If the named item has the filename set, the contents of this file are included in the mail message. The data flow can be seen visually as the bottom arrows of the plot, data is “pulled” from its origin to where it is used.

email address address

This is the email address that the email generated by this item will be sent to. This address can either be set to a single-user address, an alias or to a space-separated set of email addresses. e.g. single user: address = admin@mydomain.edu, list: address = admin1@domain1.edu admin2@domain2.com

file to include filename_0

The contents of the file specified by this parameter, and those of the referenced output file are mailed, each wrapped in: “\n---------- begin File %s ----------\n”, “\n---------- end File %s ----------\n”, where %s is the file name.

filename for digit 0 filename_0

This is the filename (including full pathname) for the ‘0’ digit NIST SPHERE file. From this filename all other digit filenames are permuted by substituting the last ‘0’ with ‘1’ through ‘9’, and optionally ‘11’ for a pause (see the speaker_mode parameter for details).

filename template filename_template

The characters in this field are added to the internal program data (including the speaker number) to create a unique filename. Note that the user may use the same filename template only once within an application.

filename to play (referenced) ref_name

Instead of specifying a static filename to be played, the user can reference the filename of another item. The referenced item should usually be of type record (i.e. play the newly recorded audio file) or shell command (i.e. play the file specified by the FILENAME: header in the program output). The data flow can be seen visually as the bottom arrows of the plot, the data is “pulled” from its origin to where it is used.

filename to play (static) filename_0

This parameter specifies the NIST SPHERE filename to be played.

goodbye message filename_0

This parameter can be set to a file that is played to the user before hanging up the line. It is played, of course, only if the user is still on the line and has not hung up.

initial value initial

This is the initial value for an iterate item. A set number item can reset an iterate item to its initial value by using the key phrase “reset iteration count” instead of specifying a numeric value.

jump if dtmf keys are hit jump_name

This item jumps to the named item if the following conditions are met: 1. this parameter points to a valid name 2. the abort keys parameter is not null 3. the number of digits is greater than 0 4. a dtmf key is hit during playback

jump to if no connection jump_name

This parameter controls the behavior of the item on an unsuccessful dial. If the dial attempt fails (no connection is made), control aborts to the item with this name. If no value is specified here, control advances to the next item regardless of exit status. The control flow can be seen visually by the arrow leaving the top of the dial item.

jump to if no match jump_name

If the comparison of the referenced data string and the string(s) specified in this item fails, control aborts to the item specified with this name. The control flow can be seen visually by the arrow leaving the top of the check dtmf item.

jump to item jump_name

Control is passed to an item named in this field. The control flow can be seen visually by the arrow leaving the top of the item’s icon.

jump to on error jump_name

If the shell program encounters an error (exit status of ISIP_ERROR), control aborts to an item with this name. The program can more reliably return an error status by printing the string “RETURN: $ISIP_ERROR” to stdout. The control flow can be seen visually by the arrow leaving the top of the shell item.

jump to on time out jump_name

If no valid dtmf key is hit within the time limit, control aborts to an item with this name. The control flow can be seen visually by the arrow leaving the top of the item’s icon.

key timeout wait_interval

This is the maximum amount of time that this item waits for a dtmf key to be hit. If an event does not come in this amount of time, the item aborts its action. If it has jump capabilities, it performs the jump in this situation. Note that the board waits for this length of time between each digit. It only tries to alter the control flow if no keys are hit. The control flow can be seen visually by the arrow leaving the top of the item’s icon.

list of numbers filename_0

This is a file containing a list of strings that is compared against the referenced dtmf string. If the referenced data string does not match any of the strings in this file, the comparison fails and the item jumps to the specified item. A successful match results in normal program flow to the next item.

max number of digits speaker_mode

This is the maximum number of digits to be read from the dtmf (touch tone) keypad. This is a useful parameter if the number of digits to be read from the user is known in advance. For a menu situation, setting this to ‘1’ causes only one digit to be read and then the system moves on instantly, waiting no longer for the next character.

no connection message filename_2

This parameter can be set to a file containing an audio message such as “No connection was made.” It can be played in the future if appropriate by using a play item configured to reference the dial item.

number to dial (referenced) ref_name

The telephone number is referenced from this item. If this is specified, the statically expressed telephone number specified in the dtmf parameter is ignored and instead, the system dials the dtmf string stored in the item named here. This should typically be set to

reference an item of type get dtmf or a shell command. The data flow can be seen visually as the bottom arrows of the plot, data is “pulled” from its origin to where it is used.

number to dial (static) dtmf

This is the telephone number that is dialed. This can be left NULL and the reference object can be set to dial the numbers set in another item, such as a “get dtmf” or a “shell command” item.

play abort keys abort_keys

These are various dtmf keys that make up the dtmf interrupt mask. Any of these keys being hit causes the current item to abort playback. If a key is not specified in this list, the board does not respond to it as a user input. If the max number of digits parameter is set to a non-zero value, then playback is aborted on any key-press (effectively ignoring this parameter). Instead, the abort keys parameter is used to specify the TERMINAL character for a string of digits of digits (if $n > 1$).

short circuit data (referenced) ref_name

This field is used to reference a string from another item. If the user specifies the name of an item that has a dtmf parameter set (e.g. get dtmf, shell command, play, wait for call, iterate, etc.), then the value in this item’s string container (dtmf) is copied into the data container for this get dtmf item as if it was entered on the keypad. Control is advanced immediately to the next item. This feature is used to create the common task block of prompting for and reading a string from the user’s dtmf keypad. The play item can be configured to read dtmf keys if they are hit during playback, but it does not wait after the end of the file if a key has not been hit. The get dtmf item is configured immediately following the play prompt to try to import the string read. If the user begins to input a string during the prompt, the entire string entered is copied into the data string container of this get dtmf item. Later in the application, the user need only to reference the data string held by this get dtmf item to read the data entered from either the play prompt or this get dtmf item. This string of characters commonly corresponds to dtmf keys on a touch-tone keypad, but any ASCII character (such as program arguments) can be specified. The data flow can be seen visually as the bottom arrows of the plot, data is “pulled” from its origin to where it is being used.

speaker mode speaker_mode

This field is copied directly into the recorded NIST SPHERE file. Commonly used values for this field are “read,” “prompted,” or “conversational.”

specific number dtmf

This is a specific data string that can be compared against the referenced dtmf number string. If the referenced data string does not match any the specified string, the comparison fails and the item jumps to the specified item. A successful match results in normal program flow to the next item. The control flow can be seen visually by the arrow leaving the top of the check dtmf item.

step value step

This is the step value for an iterate item. This is added to the internal count held by the iterate item (in the dtmf data string container) every time the iterate item is executed, including the first.

string to check (referenced) ref_name

This field contains the name of the item who's data is of interest in the comparison function. It is most likely an item of type get dtmf, shell command, or iterate. The data flow can be seen visually as the bottom arrows of the plot, the data is "pulled" from its origin to where it is used.

target item (referenced) ref_name

This field contains the name of the item in which data is set. It is most commonly used to specify the name of an inner loop iteration counter, to reset it outside of the nested loop. The data flow can be seen visually as the bottom arrows of the plot, data generally appears to be "pulled" from its origin to where it is used. In case of the set number item, however, data flows in the opposite direction of the arrow.

telephone number? speaker_mode

This parameter is used to tell the synthesize item to play the number as a telephone number (e.g. 1-601-555-1212 vs. 16015551212). Placing a true value (non-null) in this space causes the appropriate pauses to be inserted in the playback (the default behavior is to play a number with no breaks in the output).

terminal key abort_keys

The get dtmf item reads dtmf keys from the keypad, and concatenates all the accepted key-presses to the current string. This field is used to specify the TERMINAL character in such a string of dtmf key-presses.

value dtmf

This filed contains the value to be exported to the named item. The dtmf parameter of the named item is then altered to reflect the value specified here. Use of this item is the only way to externally affect another item's data. This item can also be used to reset an iteration item to its initial value. This can be done by either explicitly specifying the initial value in this field; or by using the key phrase "reset iteration count", in which case the iteration item will reset itself to its initial value. The latter method is preferred, keeping the reset value with the iteration item rather than this reset item.

wait time wait_interval

This is the maximum amount of time that this item waits for an event to occur. If an event does not come in during this interval, the item aborts its action. If the item has jump capabilities, it performas the jump in this situation. The control flow can be seen visually by the arrow leaving the top of the item's icon.

wait time between digits wait_interval

This is the maximum amount of time that this item waits for between dtmf keys. If an event does not come in this interval, the item aborts and waits no longer for further digits.

APPENDIX D. SIGNAL DETECTION PARAMETERS

The linkon system comes with a custom signal detector designed specifically for speech data collection. The complete detection process can be configured with user specified parameters, although the default values will usually suffice. We tried to be consistent with the standard naming in the DSP and speech research community, in naming the signal detector parameters. Figure D.1 shows a typical speech signal and some important parameters which are described below. All signal detection parameters related to time are specified in seconds.

It should also be noted here that in conference record mode cropping is used to guarantee exactly equal duration files on the master and slave channels.

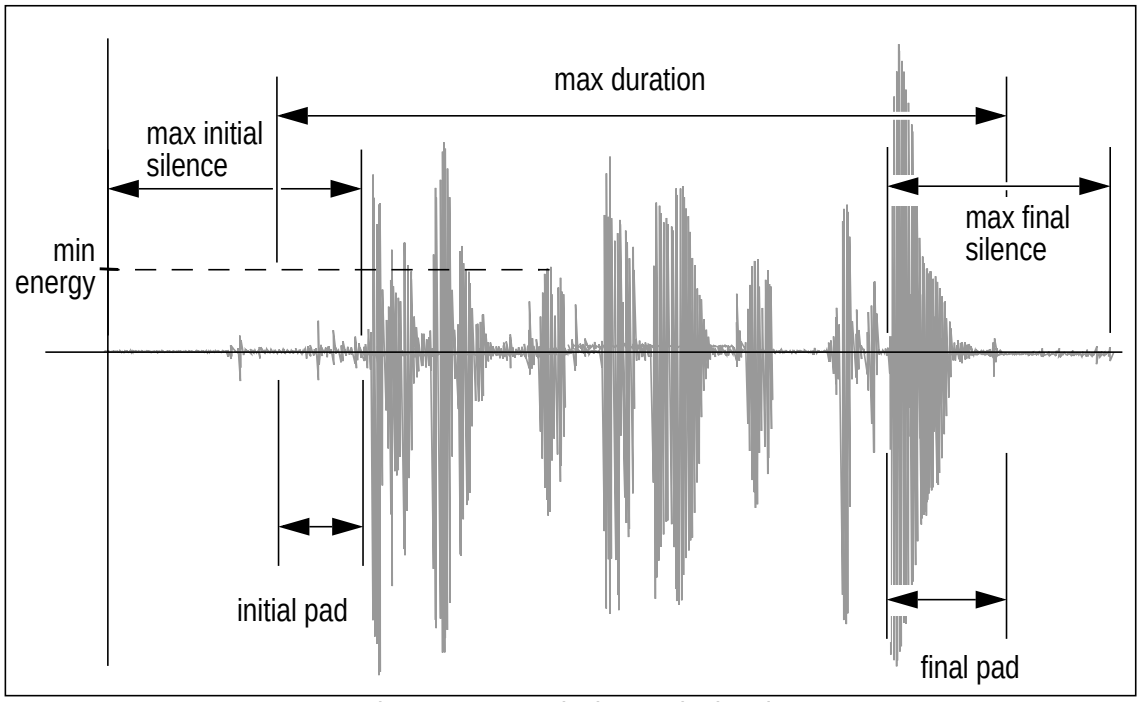


Figure D.1: A typical speech signal.

non-signal-detected time wait_interval

Using this mode for record disables the signal detector. Only a specified abort key or recording time exceeding a preset value will end the recording process.

sample frequency sample_frequency

This is the sample frequency at which the data will be sampled during recording. Currently only 8000 Hz is supported by the Linkon hardware.

initial pad time initial_pad

This is the time in the recording process prior to which the signal detector is inactive. This is useful, since in many instances the signal detector crops the initial part of the of the speech because of the low signal levels.

final pad time final_pad

This amount of time (in seconds) will be padded to the end of the recorded utterance. For the board's signal detector this is a particularly necessary parameter, since the detector only registers a rising edge of an utterance, a sustained signal will not be considered. As a rule of thumb, for a symmetrical utterance, set this parameter to approximately 0.5 seconds longer than the initial_pad parameter.

maximum initial silence max_initial_silence

This is the maximum amount of silence that the signal detector will wait through before it detects the first signal. If a silence longer than this period occurs before the first signal, no utterance will be recorded.

maximum final silence max_final_silence

This is the maximum amount of silence that the signal detector will wait through after already having heard a signal. Another way to think of this parameter is the maximum silence that will occur between linguistic entities (could be words or syllables) in utterances.

maximum duration max_duration

This is the maximum duration of time that you wish an utterance to be recorded. After this time has expired, recording will terminate and the file will be cropped. The total time of the recorded disk file will include the maximum duration and both the initial and final pad times.

minimum detection energy min_energy

This parameter allows the sensitivity of the signal detector to be calibrated about the channel's noise level. 500000.0 is a reasonable value to use for a noisy analog line, a digital line will have a much lower value. Having this value set too high will cause speech signals to not register, too low will cause noise to register to the detector.