XMESS: A GRAPHICAL VOICE-MAIL INTERFACE

by

Lorne David Berman

SUBMITTED TO THE DEPARTMENT OF PHYSICS IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

BACHELOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1990

Signature of Author _____ 17 MAY 90
Department of Physics
June 4, 1990

Certified by _____ 17 May 90
Christopher Schmandt
Thesis Supervisor

Accepted by _____
Professor Aron Bernstein
Chairman, Undergraduate Thesis Committee

XMESS: A GRAPHICAL VOICE-MAIL INTERFACE

by

Lorne David Berman

Submitted to the Department of Physics on June 4, 1990 in partial fulfillment of the requirements for the degree of Bachelor of Science.

Abstract

This paper describes *XMess*, an *X-windows* based application that controls a graphical user interface to an interactive voice-mail system. The audio interface and voice-mail server are handled by a separate process, *Phone Slave*. The interaction between these two processes provides the user with a complete voice mail system which is accessable from any workstation capable of running *X-windows*. Additionally, *XMess* is compatible with other "Conversational Desktop" applications, such as an electronic rolodex (*XRolo*), and a telephone interface (*XPhone*).

Thesis Supervisor: Christopher Schmandt
Title: Principal Research Scientist

Table of Contents

List of Figures

Acknowlegements

This work has been made possible by a grant from Sun Microsystems Inc.

And Geek, thanks.

1. Introduction

1.1 The Original Phone Slave

XMess[1] is an application designed to replicate some of functionality found in the original *Phone Slave*. The original project consisted of an audio and graphical interface to a telephone messaging system. The audio component was much like the present day voice-mail systems, but with somewhat more functionality. For example, *Phone Slave* was more than a simple telephone answering machine with regard to its message taking abilities. *Phone Slave* would ask a caller a series of questions, such as:

- Who's calling?
- What's this in reference to?
- At what number can he reach you?
- When will you be there?
- Can I take a longer message?

At sometime during this exchange, an actual outgoing message recorded by the owner would be played.

This message sequence served two purposes. It encouraged the caller to leave a message, and it allowed the owner of the system to access specific information, such as the caller's phone number, quickly. In addition, *Phone Slave* recognized a number of known callers who were identified by of voice recognition. These people were not subjected to the *Phone Slave's* normal message sequence. Instead, they were able to send and receive personalized messages [1].

To gain access to the message recording and playback system, the owner had the

---

[1] *XMess*: *X-windows* telephone *Messaging* system

option of calling the *Phone Slave* and using the audio interface, or sitting at his terminal and using the graphical interface. Through the use of a touch-sensitve display, message segments could be played by touching a ''sound bar''--an iconic representation of the duration of a sound. By touching the screen, one could play entire messages or groups of message segments (e.g. all of the ''name'' message segments).

Besides message recording and playback, the graphical interface included access to a rolodex and to the telephone. Each known user had an associated rolodex card which contained information such as the the address and telephone number. *Phone Slave* could serve as an auto-dialer, dialing by a rolodex name, or dialing through normal keypad-type input. In addition, *Phone Slave* had some other features, such as electronic mail access, which will not be discussed.

1.2 The Conversational Desktop

XMess is an application designed to perform as a graphical interface to a new implementation of the *Phone Slave* project. This new system is actually a series of independent applications, which together, provide similar functionality to the original *Phone Slave* system. Collectively, the programs are part of the ''Conversational Desktop''--a concept of integrating audio into the workstation environment. The four programs which constitute some part of the ''Phone Slave system'' are as follows:

- **Phone Slave** - a term now reserved for the audio interface messaging system.
- **XMess** - an application that uses a graphical interface to display *Phone Slave* messages.
- **XRolo** - an independent rolodex system.
- **XPhone** - an auto-dialer. Features include dial by name, number, and pull-down menu [2].

There are reasons why the new Phone Slave system is separated into several

applications, not the least of which is the ease of development. Each application could be programmed independently, as long as each could be integrated at a later time. The original *Phone Slave* project was one large process, which made it harder to maintain and debug, while the new system's individual applications are smaller and easier to manage.

Because of the need for each of the processes to communicate, a well-defined communications protocol was developed and adopted by each application. Since the code for this task now exists, the integration of new applications to the Conversational Desktop environment becomes almost trivial.

Additionally, some of the applications, such as *XRolo* and *XPhone*, are noteworthy

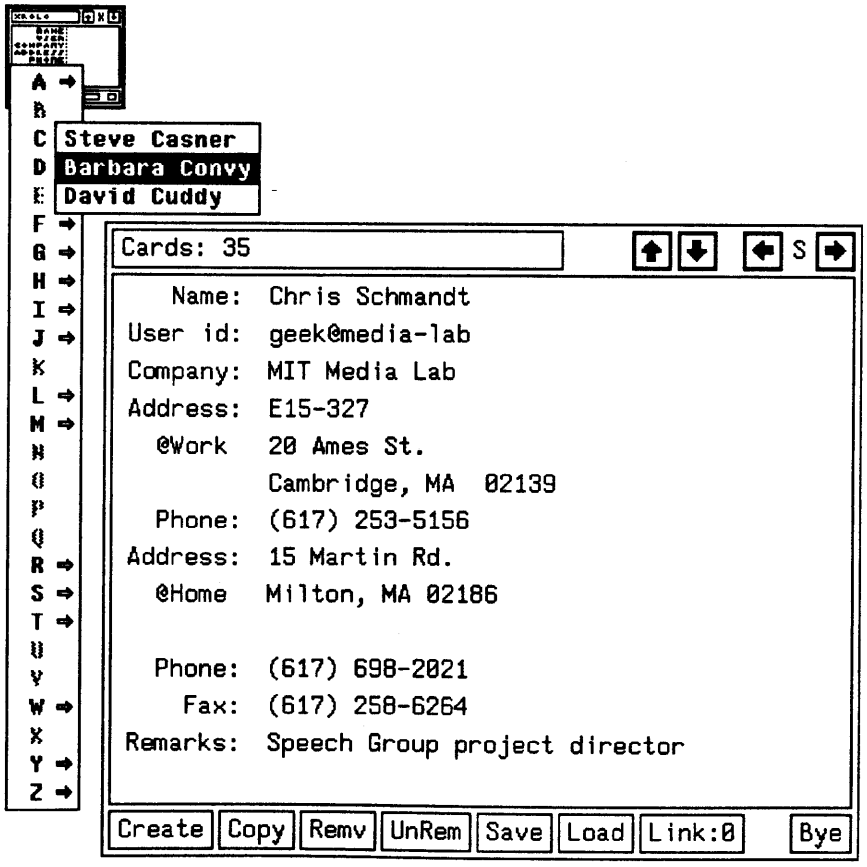

Figure 1-1: *XRolo* - A pull-down menu is used to recall another card.

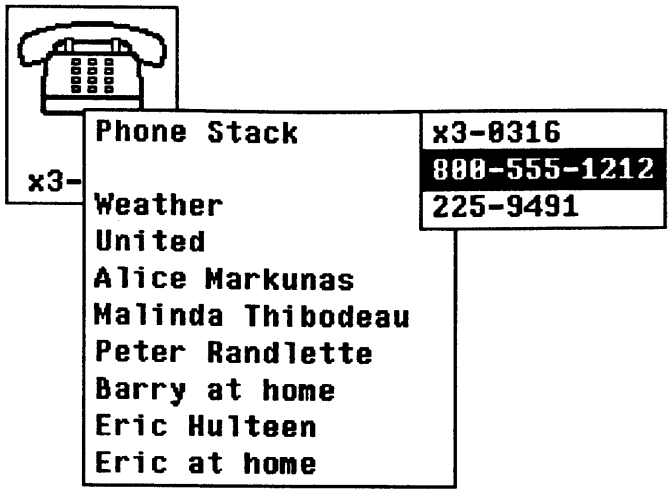

Figure 1-2: *XPhone* - Cascading menus allow several methods of dialing.

in their own right. As complete, independent programs, they may be used on workstations that do not have the capability to run the entire "Phone Slave" environment. The only requirement is for the workstation to run *X-windows*, which is widely available on a variety of different machine architectures.

Perhaps most important, a system is simply *easier* to use when separated into small, easily manageable components. The screen becomes less "cluttered" and access time is reduced. For example, in order to paste a known user's address into a document, a user would not have to access the entire telephone messaging system--a click on the *XRolo* pull-down menu would suffice. In other words, each application could develop its own efficient graphical user interface.

1.3 XMess and the New Phone Slave

Phone Slave provides a messaging system and an audio interface similar to the original version. Callers are still greeted with a series of questions, and the recorded answers are stored in independently accessable sound files. The owner and known caller functions are also implemented, using keypad instead of voice recognition.

All data that *XMess* and *Phone Slave* generate and use are stored in a number of databases. These databases are managed by a network wide server, *Netdb*, which allows multiple processes to access the information simultaneously [2]. In this setup, three databases are shared--one each for incoming calls, outgoing messages and known users. *XRolo* also has access to the known users database which it uses to display any rolodex cards upon the request of the *XMess* user. In addition, when *XMess* and *Phone Slave* are running simultaneously, they employ a method of realtime communication through the network using the *X-windows* selection mechanism.

XMess uses a system of windows to display the various types of incoming and outgoing calls. All input is done via mouse and keyboard, which is different than the touch screen interface used by the previous version. It seems in recent years the mouse has gained almost universal acceptance as the primary method of controlling window systems.

The top-level window allows the user to access the incoming messages in much the same way as the original implementation. Calls are stacked vertically, with their individual message segments arranged horizontally in columns. The sound segments are represented by ''sound viewers'', which offer significantly more flexibility than the ''sound bars'' previously mentioned. A sound may be started, stopped and even scanned with a few simple mouse motions. These sound viewers are to be used in *all* Conversational Desktop applications whenever a sound is to be displayed. Another feature of the sound viewer is its ability to be ''selected'', so that sounds may be moved between different applications.

Each column in top-level window has a header, such as ''Name'' or ''Phone Number'', which may be clicked on to play the column's message segments. The user may also click on the call header located to the call's left, to play an entire message.

Finally, each segment may be positioned and played under the control of the mouse.

Another window is dedicated to known user functions. Users may be added, deleted or have their logon IDs changed. The owner may record and playback any of the personalized messages he wishes to send. Similarly, a different window contains a list of the outgoing messages that may be played to a caller. The current message may be changed to any sound segment found in the list, or a new message may be recorded.

When *Phone Slave* in running, a network message is sent to *XMess*, which in turn, informs the user that "Phone Slave is answering an incoming call..." If the caller identifies herself as a known user, *XMess* may display "Malinda is on the phone." In either case, the user has several options. He may listen to the call-in-progress by switching on the telephone speaker (which may be done in *XMess*), ignore the call, or interrupt *Phone Slave* and handle the call manually.

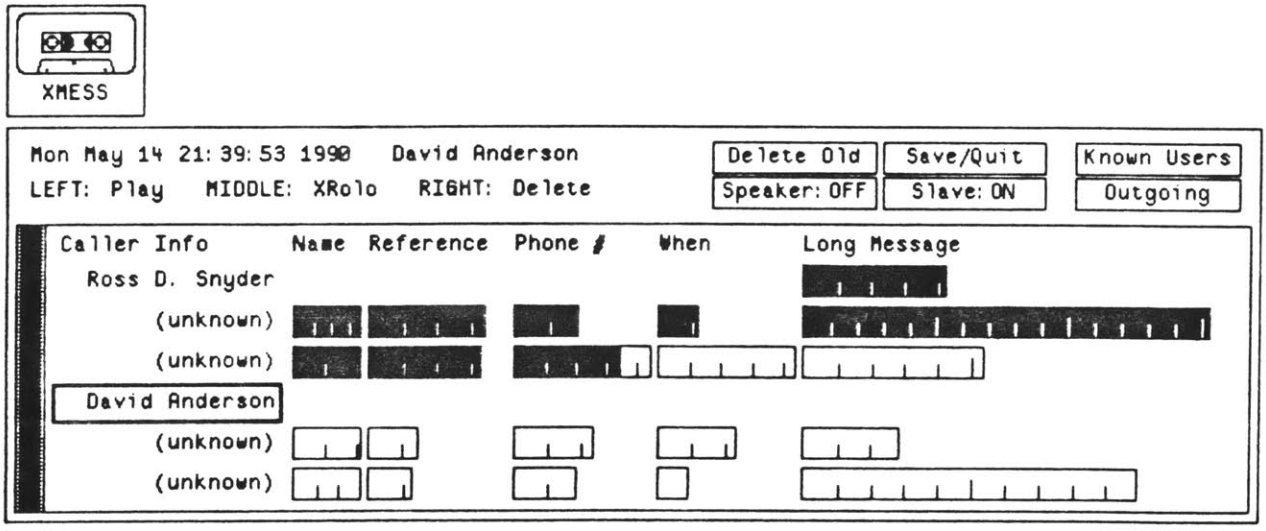

Figure 1-3: *XMess* Main Window - Incoming messages are being played.

1.4 Documentation Layout

The remainder of this paper is concerned with *XMess* documentation. Chapter two describes *XMess* operation some detail, but there is no real substitute for just sitting down and actually tinkering with the program. Before attempting this, however, the user must read chapter five to become familiar with the variety of databases that are used. Chapters four and six may be read at leisure, or when a problem arises with either the sound server or *Phone Slave* communication. The programming details of the *XMess* window functions are located in chapter three, which should only be read by those wishing to alter actual code. Since *XMess* is only an interface to *Phone Slave*, however, perhaps the best place to start is with that application's documentation [2].

2. XMess Instructions

2.1 Startup

Before starting *XMess*, the netdb server must be running. This server allows *XMess* to load and share the three databases that it uses. Sections 4.3, 4.4. and 4.5 discuss these databases in detail. It is recommended that the user become familiar with their respective formats before attempting to use *XMess*.

Since the purpose of *XMess* is to display a graphical representation of *Phone Slave* messages, a calls database constructed by that application is necessary. Additionally, to understand the full functionality of *XMess*, *Phone Slave* should be run concurrently. The instructions for *Phone Slave* may be found in the document *Phone Slave II: A Modular, Portable Reimplementation* [3].

2.2 Main Functions

On startup, *XMess* displays its tape icon. The icon serves as a "message waiting light" and is also used to toggle the popup state of the main window. Upon the arrival of new messages, if the main window is closed, the icon flashes. Clicking on the icon opens (or closes) the main window.

The main window has several smaller window areas. In the upper left corner, call information is displayed. Directly below is another message area which displays basic status information, such as "Please wait..." On the right is a layout of command buttons with various functions, and below all of this is the call window. When *Phone Slave* is not running, some of the command buttons (Such as "Speaker:ON/OFF") are "dimmed" to show their state of inactiveness.

2.2.1 Accessing Messages

The call window contains a vertical list of the incoming messages. The top line of this window contains a series of headers corresponding to each segment of a message. Clicking on a header causes *XMess* to play every sound segment in that column. The only exception is the first header, ''Caller Info'', which when clicked, plays each and every message in its entirety.

Each call is made up of a call button, located on the left, and between zero and five sound viewers following on the right. The call button is labeled with the name of caller who left the message, or ''unknown'' if this information was not available (i.e. the caller was not a known user). When the mouse pointer is placed on a call button, the date, time and status of the message is displayed in the call information window. The status field may contain an ''N'', ''U'' or a blank. New messages are represented by the ''N'', while unread messages--those that existed since the last invocation of *XMess* or *Phone Slave*--are depicted by the ''U''. Otherwise, the field is left blank.

There are three ways the user can play a message. First, the header buttons may be used. This method, however, plays the entire series of messages. Second, the user may click the *left* mouse button on the call button. The message's list of sound segments is then played in sequence. Third, an individual segment may be played through the use of the sound viewer. Every sound viewer in *XMess* is accessed in the same fashion; The *left* button starts and stops the sound, while the *middle* button is used for positioning.

Anytime a sound is played, *XMess* must access the sound server. The sound server is also used by *Phone Slave*, therefore, the two applications must negotiate for its control. The ''Slave:ON/OFF'' command button reveals which one of the applications is currently in control. If the label reads ''Slave:ON'', then *Phone Slave* may access the server and take incoming calls. However, if *Phone Slave* is ''OFF'', then incoming calls

are not answered. In this situation, *XMess* may access the sound server and play messages. *Phone Slave* may be manually turned on and off through the use of the command button, or *XMess* may perform this task automatically whenever the user requests that a sound be played.[2]

Messages may also be deleted in three ways: Clicking the *right* mouse button on a call button removes a single message. Pressing on the ''Delete Played'' command button removes all read messages. Finally, deleting messages using the *Phone Slave* interface will trigger their removal in *XMess*.

There also exists the option to popup the rolodex card corresponding to the label on the call button. Assuming that *XRolo* is running and using the same database as *XMess*, clicking the *middle* mouse button on the call button will perform this rolodex action.

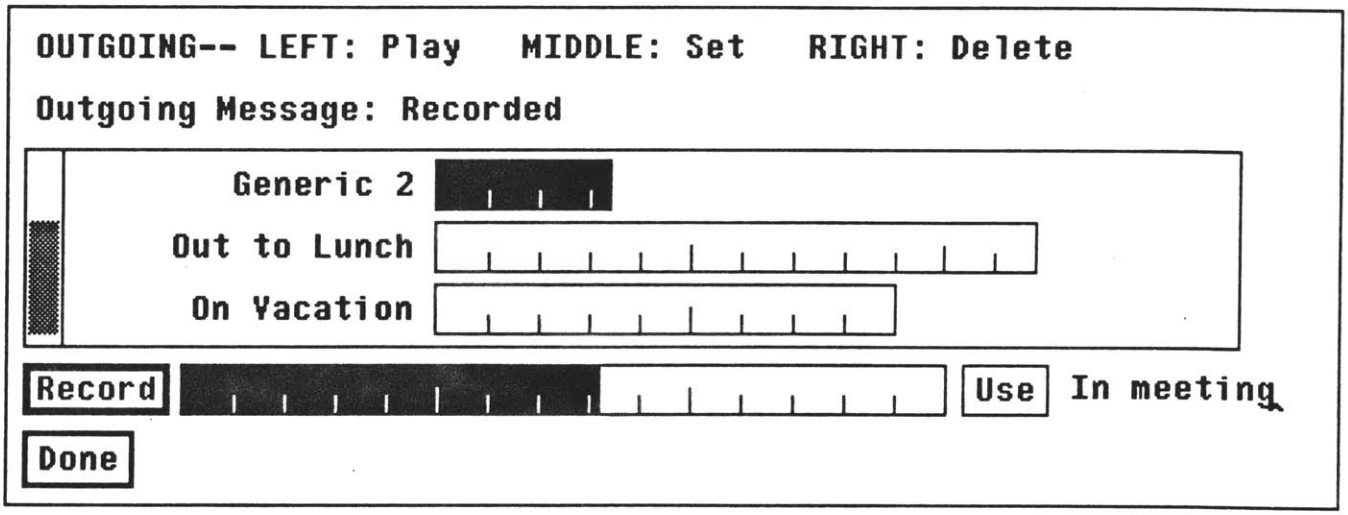

Figure 2-1: *XMess* Outgoing Message Window - A new message is being recorded.

[2]This resource arbitration would be better done in an audio server such as that described in [4].

2.2.2 Selecting Outgoing Messages

In addition to playing the incoming messages that were recorded by *Phone Slave*, the user has the option to select, delete, or record a new outgoing message. These functions are supported in a popup window that appears when the user clicks on the "Outgoing Messages" command button.

The popup contains two windows. The top window holds the list of all the available outgoing messages. These messages, similar to the incoming calls, are arranged in message button/sound viewer pairs. The message button is labeled with the name of that particular message, while the name of the current outgoing message is printed at the top of the window. Clicking the *left* mouse button on a message label plays the message (which also may be played using the standard sound viewer actions), and the *middle* mouse button selects a new outgoing message. Except for the first, the *right* mouse button deletes any unwanted messages. The first message is reserved for *Phone Slave* remote recording, and must always exist.

To record a new outgoing message, the user presses the "Record" command button and begins to speak. When finished, the user may add the recording to the outgoing message list by pressing "Use". Otherwise, the message may be re-recorded or aborted. The message button obtains its label from the text area beside the "Record" button. The user may wish to alter the default text that appears in that area *before* the message is appended to the list.

2.2.3 Known User Functions

To open the known user popup, the user clicks on the "Known Users" command button. This popup allows the user to add, delete and record messages for known users.

To access a (potential) known user's status, his name (full name, username,

company, or a substring of any of these) is typed into the text field located to the right of "Name". For a name to be valid, it must *already exist in the known users database* which should be modifiable by *XRolo*. The existence of a name does not imply that the person is recognized as a known user. That status is reserved for those who have a valid telephone ID. The ID is displayed in the text area directly below the "Name" button. A known user may be added, deleted (by simply removing the ID) or have her ID changed by altering the text in the ID window.

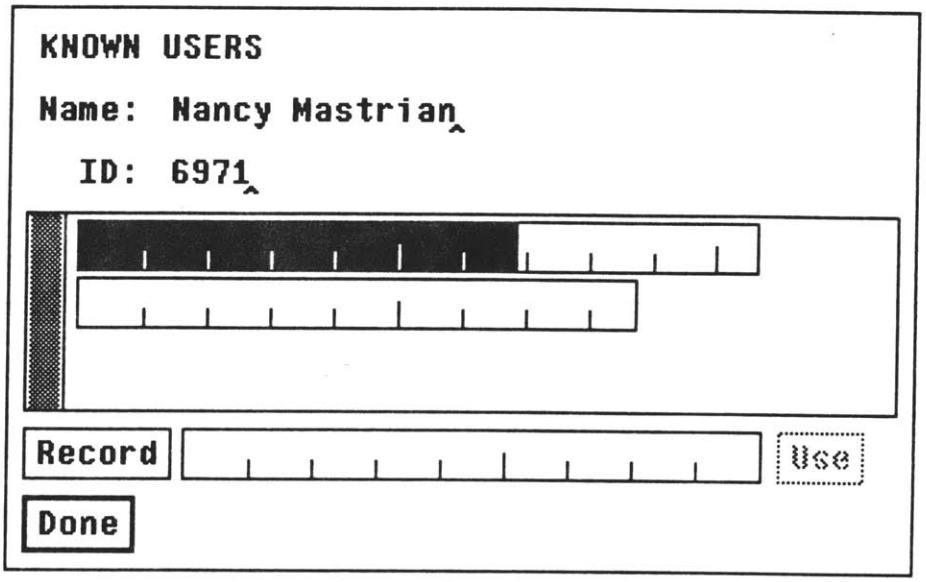

Figure 2-2: *XMess* Known User Window - Sending personalized messages

A list of the known user's pending messages is found in the window below her known user name and ID. To play these messages, the user must perform the appropriate sound viewer actions. The recording of an additional personalized message is similar to the outgoing message procedure. Since these messages do not have any associated buttons, however, there exists no text area to label the message. All the messages for a

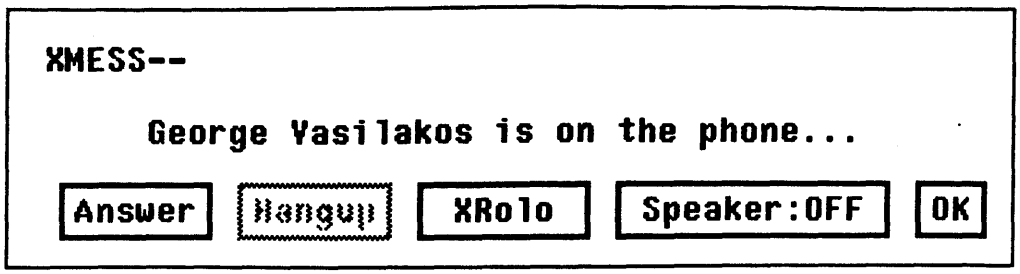

Figure 2-3: *XMess* and *Phone Slave* allow the monitoring of incoming calls.

particular user are considered to be a group and functions such as "delete message" are not provided.

2.2.4 Realtime Call Functions

When *Phone Slave* detects an incoming call, it notifies *XMess*. *XMess* then pops up a window, informing the user of this development. If the caller decides to identify himself as a known user, *XMess* displays the caller's name.

At the completion of the call, the window is removed and *XMess* is returned to its former state. The user may decide to remove the window manually by clicking on "OK". However, since *Phone Slave* uses the sound server to record the call, until the call is completed, the user is prohibited from performing any action that disturbs the server--e.g. playing any sound.

Otherwise, the user has a myriad of choices concerning the call. The speakerphone may be toggled with the command button "Speaker:ON/OFF". Note that this button is duplicated in the main window. If the caller is a known user, the appropriate rolodex card may be made to appear with the "XRolo" command button.

These previous functions are transparent to the caller. The user, however, may wish to interrupt the *Phone Slave* process and handle the call directly himself. In this case, the command button "Answer" is used. *Phone Slave* halts any process in which it was

engaged, and allows the user to handle the call manually. Upon completion, the user places the phone on-hook and presses the "Hangup" button to let both applications know that the call has ended.

3. XMess Windows

3.1 Basic Setup

XMess is composed of five major windows. The first window appears on startup. The only feature of this window is the *XMess* icon that it displays. Upon a left button click, the second window, which is the main display for incoming messages and button commands, is toggled. The third and fourth windows appear upon the mouse selection of the command buttons ''Known Users'' and ''Outgoing Messages'' which are located in the main window. The final window appears when an incoming call is detected. This window can not be made to appear manually by the user.

3.2 Icon Window

The icon window, titled `shell` is the major parent window. To display the icon, a label (widget type *labelWidgetClass*) is created inside a form (type *formWidgetClass*). This label, as with all other *XMess* labels, is created with an argument list `label_args`. This argument list, along with other arglists, is defined globally at the beginning of *XMess*. The form widget is defined by `top_form_args`, as opposed to the other form arglist `form_args`. The major difference between the two is that `top_form_args` allows the form to be resized, as opposed to `form_args` which does not. The arglist `top_form_args` should be used in most ordinary situations, while `form_args` is designed to be used inside a viewport. The *XMess* tape icon is a bitmap that is defined in the include file ''XMess_tape.bm''. There is also a blank icon, ''XMess_alt.bm'', that is sometimes periodically switched with the tape icon, giving the illusion of a blinking tape (which informs the user of new pending messages). The blinking is accomplished through the use of an X interval timer and the callback `flashtime_proc()` which toggles the icon. Finally, the capture of a left button press is done through the use of a

translation table and an action table. These tables, along with the other tables, are defined at the top of *XMess*.

3.3 Main Display Window

The main window is actually a popup shell of type *transientShellWidgetClass*. The window is displayed through the use of `XtPopup()` with an argument of `XtGrabNone` (to allow the icon window to receive events). The popup shell has only one child, namely `bigform`. `Bigform` is the parent to `topform` and `view`. `Topform` is the form that contains the main *XMess* control panel. Inside is a mass of widgets, consisting of two label widgets and six command widgets. The label widgets are used to display status information and warning messages. Note that these widgets are fifty characters wide. The character used, however, is not a space, ' ', but rather the letter 'L'. This is to ensure that in the case of a proportional font, the label size would not be based on the smallest character in the set (space) but based on an averaged sized letter ('L'). The command buttons are all set to the same width for the sake of aesthetics. The width is determined by the width of the widest button, ''Speaker:OFF''. The command buttons ''Speaker:OFF'' and ''Slave:OFF'' are only activated (sensitized) by `fix_slave_wids()` if Phoneslave is connected.

Below `Topform` is `view`, a viewport that encompasses the widgets which compose incoming messages. `View` has a child `form` which actually parents these widgets. A viewport is used because the number of messages, hence the number of widgets, to be contained in `form` is unknown. The viewport allows the scrolling of the form through the use of a scrollbar. Therefore, `view` and the visible portion of `form` can remain fixed in size while messages are added and deleted.

The top of `form` is composed of a series of command widgets. These widgets are

meant to title each possible segment of a message. When clicked on, the column of sounds under the widget is played. Their titles are contained in a character array, `*title_label[MAX_HORIZ_SV+1]`, which is located at the top of *XMess*. `MAX_HORIZ_SV` is currently defined to be 5, the maximum number of message segments. The '+1' allocates storage for the first command button, which is not associated with any particular sound segment, but is needed to title the command buttons that accompany each message.

Below the titles is a series of vertically stacked incoming calls or messages. Each call is composed of a series of horizontally arranged widgets. The leftmost widget is a command widget, which labels the call with the name of the caller. To the right, there are between 1 and MAX_HORIZ_SV SoundViewer widgets, each in the appropriate column designated by the title widgets.

3.3.1 Creating Call Widgets

To create the widgets needed to display a call, one uses:

```
create_but_and_sv(bl, date, callnum,
                  names, lens, under, num)
  char *bl;
  char *date;
  int callnum;
  char names[MAX_HORIZ_SV][MAX_SOUND_NAME_LEN];
  int lens[];
  int under[];
  int num;
```

The command button associated with each call is given the label `*bl`. To display the time and date of the call (in the message widget), each call also stores a string, `*date`, which contains this information. The `callnum` is an integer which is a handle to the number of the call as referenced in the calls database. Each SoundViewer must have a

sound filename which is passed in by the array `names[][]`, and each sound file has a length (in msec) which is passed in by `lens[]`. Since a call may have between 1 and MAX_HORIZ_SV sound segments associated with it, `num` contains the actual count. Finally, the array `under[]` contains the column at which to place the SoundViewer. Normally, there are five (MAX_HORIZ_SV) columns and five sound segments for a call. The array under, therefore, should be `{0, 1, 2, 3, 4}`, with each column receiving a sound segment. Alternatively, a column number may be set to -1, telling *XMess* to place the sound segment in the next available column. To place five segments, one can also use an array set to `{-1, -1, -1, -1, -1}`. For this purpose, a global array `std_under[]` is defined as such.

For the purposes of the window system, the call widgets (which are not actual widgets) are all referenced by a button number. This number is defined as the element of the array `(Widget)but[]` in which the command button is located. Each array element of `but[]` is initialized to NULL, which indicates its unused state. A subroutine `(int)get_next_free_but()` returns the number of the next available but. This number is used for all references to any widget associated with that call. For example each button has two other button numbers associated with it: `am_above_but[]` and `am_below_but[]`. This doubly-linked list allows the easy deletion and fast search time of calls. Since `create_but_and_sv()` only *creates* buttons (it does not *place* them), `am_above_but[] = am_below_but[] = -2` to show that the button is not yet placed.

The allocation of memory for sound segments is similar to that of the command buttons. There is an array `(Widget)sv[]` to store sound widgets, and a subroutine `(int)get_next_free_sv()` to return the next available sv. The integer returned, however, is *not* a handle to the command button. To find the button number of an SV

(and find the handle to the entire call), reference `(int)my_parent_but[`*sv_number*`]` which is set inside `create_but_and_sv()`.

Both the button widget and SV widget are created with `XtCreateWidget()` as opposed to `XtCreateManagedWidget()`. This is because `create_but_and_sv()` only creates the widgets and returns a handle (the button number). Creating a managed widget automatically maps that widget. This is undesirable, because when a large number of widgets is displayed, it is better to format them *after* they have all been created instead of displaying and *then* formatting them. The latter method makes for a messy display.

The button widget is modified with the standard `but_args`, while the SV widget uses (appropriately) `sv_args`. The resource arguments that need to be modified for each SV are as follows: *XtNwidth*, *XtNwidgetDuration*, and *XtNsegmentDuration*. *XtNwidgetDuration* is the length of the entire SV in milliseconds, while *XtNsegmentDuration* indicates the length of the sound (in msec) that is to be displayed in the widget. Throughout *XMess*, these two resources are considered equivalent, for it is desired to have the length of the widget entirely filled with its associated sound (with no blankspace). *XtNwidth* sets the SV's actual length in pixels. *XMess* calculates the width of a call segment through the use of the arrays `(int)max_sv_time[]` and `(int)max_sv_width[]`. The array `max_sv_time[]` is set with empirical values of what constitutes the maximum sound length that a segment will encounter. A segment with that sound length will be sized to the pixel length given by the array `max_sv_width[]`. Shorter sound lengths will be sized as a fraction of the maximum pixel length. For example, if `max_sv_time[0] = 2000` and `max_sv_width[0] = 100`, the first sound segment of a call, if it were 1.5 seconds long, would be have a widget length of 1500/2000 * 100 pixels. The defined constant `MIN_SV_WIDTH`

prevents a widget to be sized too small. In the event that a sound in longer than the maximum time, the widget is sized to the maximum length and the sound is "squeezed" to fit--i.e. the time scale indicated by the SV's tic marks is lengthened.

3.3.2 Placing and Manipulating Call Widgets

As mentioned, `create_but_and_sv()` only creates an instance of a call widget. The placement and mapping of the call widgets is handled independently. The subroutine `place_but_last(but_num)` takes as an argument the button number returned from `create_but_and_sv()`. This routine steps through the doubly-linked list of button widgets (i.e. call widgets) finding the last one. Through the use of the resource *XtNfromVert*, the button is installed in the proper place in the form, and the linked-list (`am_above_but[]` and `am_below_but[]`) is updated. In this release of *XMess*, all call widgets are placed sequentially, with the latest calls placed below existing widgets. The routine `place_but_first()`, therefore, is currently unused button supplied in the event that future versions will have need of its function.

These placing functions only position the call widgets inside a form. After placement, the widget is still not necessarily visible. Newly created widgets are created but not managed by `create_but_and_sv()`. The function `manage_all()` cycles through all the call widgets (comprised of the buttons and SVs) and manages (which maps) any that are unmanaged. This allows *XMess* to create and place a number of call widgets without the accompanying screen clutter. A call to `manage_all()` then displays them almost simultaneously.

The last call widget manipulator function defined is `kill_but(but_num)`. Passed a button number, `kill_but()` removes the call widget from the linked-list and then resets *XtNfromVert* for any widget that my be below. The form will automatically

redisplay the widgets, filling the gap left by the removed widget. Finally, `XtDestroyWidget()` is called, freeing the memory and resources used by the widgets.

3.4 Known User Window

Through the use of `XtPopup()`, the known user window appears when the user left mouse clicks on the "Known User" command button. The window is comprised of a top level form, `known_form`, and several widget children. These children are the text labels, "Name:" and "ID:", along with their respective text input widgets, the "Done" command button, and `known_view`, a viewport which allows the display of multiple SVs without resize.

Both of the text input areas are of type *asciiStringWidgetClass* and are modified with the translation tables `textTranslations` and `tidTranslations`. Carriage returns and linefeeds are used to pass control to callbacks, which allow *XMess* to act on the text input. In this case, the "Name" text callback loads the ID and any current messages associated with that known user, while the "ID" callback updates the known user ID of the most recent name searched. The translations are also modified to allow selection stuffing on button two. The "stuffing" is preceded by the erasure of any current text in the field, and followed by the same callback used for a carriage return. This allows the one button retrieval of both a known user's ID, and his/her messages. Finally, in `tidTranslations` a leave window event is also bound to the carriage return callback. This is to ensure that a known user's ID is always updated, even when the user neglects to perform a carriage return.

The carriage return callback retrieves any sound names and sound lengths associated with the known user's name. This function then calls

```
display_ku_svs(basename, arrint)
```

```
char *basename;
int *arrint;
```

where basename is the common soundname shared between each of the user's messages, and arrint is an array of sound lengths--each sound length corresponding to one pending message.

The display of known user SVs is similar to the display of call SVs. Known user SVs are stored in a widget array, known_sv[]. The function display_ku_svs() first checks to see if any SVs are displayed (i.e. if any element in known_sv[] is non-NULL), and if they are, destroys them. Destroying a KU widget is not absolutely necessary. In theory, it is possible to reuse widgets through the judicious use of XtManageChild() and XtUnmanageWidget() while updating the widgets' resource lists. Though in practice, the speed at which the XtDestroy() function operates makes it quite unnecessary to program the extra code needed to perform the task of juggling widgets.

The label widget known_sv_label, however, does use this technique. This widget is set to post the notice "No pending messages" in the SV window when there are no SVs to display (i.e. display_ku_svs(NULL, NULL)). The message is toggled with the functions XtManageChild() and XtUnmanageChild().

The remainder of the code mimics create_but_and_sv(). The function creates a series of SVs, each with a length dictated by *arrint. This length is calculated with the same formula used in creating the call widgets. In this case, the reference lengths used are the *last* values in max_sv_width[] and max_sv_time[]. The same SV callbacks, *XtNstartSoundCallback* and *XtNfinishedCallback*, are attached to each widget.

3.5 Outgoing Message Window

The display of outgoing messages in a popup window is much the same as the display of known user messages, although not nearly as elegant. The creation, setup and display of all the outgoing message widgets is handled in `display_motd_svs()`. This function digs directly into the *motd* database, extracting the sound names, lengths and titles of each outgoing message. From this information, it builds an SV and command widget for each message. The length of the SV is determined by the usual formula, with the maximum time and length (width) defined as `MAX_MOTD_TIME` and `MAX_MOTD_WIDTH`.

The translations and callbacks for the SVs are set to their normal values, while the translations for their associated command widgets are somewhat altered: the left button simply plays the entire SV, the middle button sets the outgoing message, and the right button deletes the message.

When deleting, `but_delete_motd()` first verifies that the message is not one of the defaults. If not, the database entry for that message is removed, and the remainder of the messages are renumbered (that is, each message is numbered contiguously, message 0, 1, 2, ...). Finally, *all* the widgets are destroyed then redisplayed, using `display_motd_svs()`. This is not the most elegant method of deleting a widget, (e.g. note the method used in the deletion of a call widget), but it is effective when deletions are infrequent.

3.6 Incoming Call Window

When *Phone Slave* notifies *XMess* that it is taking an incoming call, *XMess* displays a popup informing the user. The user must attend to the popup, for it grabs all input using *XtGrabExclusive* and effectively blocks the user from performing any action on the

main window. Initially, the popup notifies the user that ''Phone Slave is answering an incoming call''. In response, the user may: ignore the message (and wait for the call to end), remove the message, or answer the call. If the message is removed by selecting the ''OK'' command button, the user may again resume most normal operations except the playing of an SV. The variable `slave_busy` is set `True` which disables any SV actions. If the user decides to answer the call with the ''answer'' command button, the answer button is desensitized and the hangup button is sensitized. Selection of the ''Hangup'' button ends the call and removes the window. Additionally, *XMess* might inform the user that ''<name> is on the phone.'' In this case, the ''XRolo'' button is sensitized and the user may popup the rolodex card (assuming that *XRolo* is running) that corresponds to the person on the phone.

4. Interprocess Communication

4.1 Why Ethernet?

The sending of data from one process to another on a remote machine may be done in several ways. One method employed by *XMess* and Phone Slave is the use of common databases. This method, however, has one major drawback--namely, it does not allow for asynchronous communication. To receive a message, a process would be forced to timeout often and examine the database, which is a timely task. A more direct approach would be the use of a serial line between the two computers. The obvious problem with this method is that the serial line must *exist*, which is not always the case. The solution, therefore, is to empoly ethernet, which is a system that supported in hardware and software by all of our machines. The software exists on all Unix-based machines, allowing the communication between any two machines that share the net.

4.2 Establishing a Connection

The software for establishing a communications link between *XMess* and *Phone Slave* is located in "inet_comm.c", with a supporting header file "soccomm.h". *Phone Slave* shares this header file with *XMess*, although it does use a slightly different source code. The code only differs in program dependent functions, but the communications software is the same.

The function `inet_open()` probes the net, attempting to find the machine that hosts *Phone Slave*. The names of the machines *XMess* searches is defined in the array `(char *)hostlist[]`. Before probing remotely, *XMess* must create a socket (i.e. an endpoint for communication). The function `socket(AF_INET, SOCK_STREAM, 0)` returns an integer, `sock`, that is used to reference the connection. The parameters passed to `socket()` define the format in which the data are to be sent, and the type of

communication allowed. `AF_INET` is the Internet standard protocol, while `SOCK_STREAM` is the communications type for two-way reliable data transfer.

The variable `sock` is a temporary handle to reference a communications port. These ports--at least with respect to *XMess* and *Phone Slave*--are hard coded into each machine. All potential hosts have a defined communications port which *XMess* and *Phone Slave* use. To grab a handle to this port, `inet_open()` uses `getservbyname("foneChris", "tcp")`. This function returns a structure which contains the port number found to belong to "foneChris". Besides the address of the port that the data is to be sent, the address of the recipient machine is also needed. The address of the target machine returned by the function `gethostbyname(hostname)`. The port address and the target machine address are then placed in the structure `slavehost`. The attempt is now made to open communications with the target machine through the use of the function `connect()`, which is passed `sock` and `slavesock`.

If the connection is successful, the socket connection is left open. Some additional work with the function `fcntl()` must be performed on the socket to ensure that data is received asynchronously. At this time, *XMess* may recieve data through the port. To avoid the issue of manually polling the socket for data, a mechanism is setup whereby *XMess* automatically receives incoming data. When data arrives, the signal `SIGIO` is activated. *XMess* traps any occurrence of this signal and automatically jumps to the procedure `inet_datain()` when it arrives.

If the connection fails, however, another attempt is made to connect to the next machine in `hostlist`. When all available machines have been probed, and a connection still does not exist, it is assumed that *Phone Slave* is not running. Nonetheless, it is important for *XMess* to listen on the socket in the event that *Phone*

Slave is started at a later time. In this case, the socket should listen to any machine willing to communicate over the "foneChris" service port. The target machine address, therefore, is set to `INADDR_ANY`. Instead of connecting to the port (which is not possible), the socket is bound to the port with the `bind()` function whose format is analogous to `connect()`. To listen for an attempt at a connection by Phone Slave, *XMess* uses `listen(sock, 1)`. When the connection is detected, *XMess* jumps to `inet_connin()` because of the signal trapping set up through `signal(SIGIO, inet_connin)`. In `inet_connin()`, the connection is completed when *XMess* calls `accept()` and retrieve the new socket address at which data may be exchanged. Finally, the normal procedures of setting the socket to receive asynchronously with `fctnl(slavesock, FASYNC)` and of trapping incoming data with `signal(SIGIO, inet_datain)` are performed.

4.3 Receiving, Sending and Parsing Data

The detection of data on the socket causes *XMess* to jump to `inet_datain()`. This function basically retrieves the data through the use of `read()` and places the contents into a string buffer. The buffer is then passed on to `inet_parse_command()`, where the appropriate action may be taken.

The messages that *XMess* expects to receive are all defined in the string array `s_commands[]`. The following sixteen commands are understood:

```
"0\nDie, please.\n"
"1\nHello, anybody home?\n"
"2\nCommand acknowledged/completed\n"
"3\nClosing connection\n"
"4\nCalls db has been modified\n"
"5\nHang up the phone\n"
"6\nStop waiting for a ring\n"
"7\nStart waiting for a ring\n"
"8\nKnown users db has changed\n"
"9\nMonitor calls\n"
```

```
"10\nStop monitoring calls\n"
"11\nIncoming call being taken\n"
"12\nIncoming call has ended\n"
"13\nPhone Slave is Idle\n"
"14\nPhone Slave is Active\n"
"15\nMotd db has been modified\n"
"16 %s\nKnown user with ID has identified himself\n"
```

The commands are referenced by their numbers as defined in "soccomm.h". See chapter six for command details.

A switch statement allows *XMess* to act on each command it receives. All sixteen commands, however, are not understood by *XMess*. Some, such as "Hang up the phone" are never meant to be received, but instead, only meant to be given. All other commands (e.g. "Calls db has been modified") are acted upon immediately.

To send a command, the function `inet_send_command()` is called with the command number. Those commands that are simple notifying messages, such as "closing connection" or "Calls db has been modified" are sent immediately with the `write()` function. Some "action" commands (numbers 5, 6, 7, 9, and 10), however, require an acknowledgment (command 2) by *Phone Slave*. The last command sent, therefore, is stored in the variable `inet_last_command`, and when the acknowledgment is received in `inet_parse_command()`, *XMess* can perform the appropriate action (e.g. notify the user that the task has been completed).

The action commands require *Phone Slave* to perform tasks that should not be interrupted by further *XMess* action requests. In the event that *XMess* desires to send an action command *before Phone Slave* has acknowledged the previous action command, the command request is placed on a queue. The queue is written and read by the functions `inet_push()` and `inet_pop()`. When *XMess* finally receives an acknowledgment, in addition to performing any actions required by

`inet_last_command`, `inet_parse_command()` retrieves and sends the next action request on the queue.

5. XMess and Phone Slave Databases

5.1 Calls, Users and Motd

Phone Slave and *XMess* both maintain three common databases: calls_db, users_db and motd_db. The calls_db contains information about the incoming messages that *Phone Slave* has recorded. The users_db, which is also used by *XRolo*, contains the list of known users, their personal outgoing messages, and a wealth of other information used by the rolodex. The list of generic outgoing messages, or messages of the day (motd), is found in motd_db.

5.2 Loading a Common Database

Normally, a database is first created through the use of `ndb_create()` and then loaded with `ndb_read_db_from_stream()`. All references to that database are made through the handle returned from `ndb_create()`. Any additions, changes, or deletions pertain only to the database with the given handle number. This normal procedure for loading a database is undesirable to any program that needs to share database information between different processes. For example, suppose both *XMess* and *Phone Slave* independently create and load a calls_db. Once loaded from disk, any operations performed on the calls_db by either of the programs would not be seen by the other. Eventually, both programs would flush their own database to disk. A subsequent load from disk would contain only the changes made by the last program to write the file. The inevitable result would be a loss of data.

The solution is to allow both programs to alter the `same` database. Only the first program to boot would actually read the database from disk. Subsequent programs would receive the handle to the database, allowing them to read and change data at will. Any of the processes may safely dump the database to disk, sure that all of the data

would be written.

The question then becomes, how do multiple processes obtain the handle to a specific database? One possibility is through the use of socket communications. For example, if *Phone Slave* boots before *XMess*, then it could send a message over the net to *XMess* that contained the calls_db handle number (which is an integer). This is a simple and efficient method for *XMess* and *Phone Slave* which already have a system of net communications. Not all database clients, however, have such a system. For example, xrolo needs to share the users_db with *XMess* and *Phone Slave*, but it does not need net communications system. It would be foolish to add another layer of complexity to the program when the need is so minimal.

The solution is to use the system that all these programs share--namely, the database manager (netdb) itself. Netdb maintains a common record whose handle is returned through `ndb_get_common_rec()`. Any application may store or retieve data from this record. By convention, the common record stores field/value pairs which allow any program to retrieve the value of any globally defined variable. For example, each database is given a unique identifier name. In the case of calls_db, the identifier is ''<name>'s calls db''. The value of this field is the handle to the database. If no such field exists, the database has not been loaded. The function

```
int get_used_db(key, dbp)
    char *key;
    DB *dbp;
```

performs this procedure. The variable `key` contains the identifier, and `dbp` is an address where the function can place the handle to a new database if none already existed.

This function makes use of the netdb command `netdb_test_set_field()`. This command expects to be passed a record, a field, and a value. If the field in the given

record exists, `netdb_test_set_field()` returns a negative number. Otherwise, the field is created and set with the given value, and the field handle is returned.

`Get_used_db()` first tries to set the field `LOCK`. The `LOCK` field is set by an application when it first gains access to a database. In the event that the database is new, the `LOCK` field prevents another application from trying to use the database until all data has been loaded. The `LOCK` mechanism does not actually disallow any access to a database. It only suggests to an application that if the field is set, the application should wait until until the field is removed.

When the lock field is eventually set, `get_used_db()` grabs a new database handle with `dbp = ndb_create()` and then calls `ndb_test_set_value()` with the parameters `ndb_get_common_rec()`, `dbp` and "<name>'s calls db". If the field already exists, the function removes the lock and returns. The application would then examine the common record and extract the database handle from the correct field. Otherwise, the application uses the handle returned in `dbp` to load in the database data and *then* it removes the lock.

5.3 Calls_db format

The calls_db maintains the record of incoming calls. Each call is a separate db record which contains such information as the date of the call, the number of the call and the filenames of the recorded messages. The following is an example of a calls_db:

```
Start of DB {
Record {
{number of calls in db} {2\00}
{highest-numbered call} {1\00}
{Header record} {}
{number of call} {0\00}
}

Record {
```

```
{date of call} {Mon Apr  9 12:29:32 1990
\00}
{status flags} {D-\00}
{type of call} {Unknown caller\00}
{segment times} {950 1568 1584 2000 6540 \00}
{sound filename prefix} {\\lberman\\calls\\call0_\00}
{call date, UNIX time} {639678572\00}
{number of call} {0\00}
}

Record {
{date of call} {Mon Apr  9 12:45:55 1990
\00}
{status flags} {-N\00}
{type of call} {Known caller message\00}
{segment times} {9600 \00}
{caller name} {Chris Schmandt\00}
{caller phone id} {2021\00}
{sound filename prefix} {\\lberman\\calls\\call1_\00}
{call date, UNIX time} {639635461\00}
{number of call} {1\00}
}
}
```

The field labels (e.g. "highest-numbered call" or "Header record") for any *Phone Slave* or *XMess* database are defined in "slave.h".

Every calls_db begins with a header record. The header record maintains the number of calls in the db and the highest call number. *Phone Slave* updates the header record for its own use--*XMess* has no need for the data.

The call records are composed of several essential fields. These fields, as shown above, must be present in the record for the call to be valid. The verification of all the records, including the header, is performed by the function `verify_calls_db()`. If any of the required fields are missing in a call record, the call is deleted. The header, if found to be inconsistant with the data, is reformed with `setup_calls_header()`.

With the exception of ''status flags'', each field is created by Phone Slave at the time of the incoming call and is not altered anytime thereafter. A description of these fields is as follows:

Each call may be referenced through its call number (stored, most appropriately, in ''number of call''). The only guarantee is that a larger call number is more recent than a smaller call number. A numeric sort on the ''number of call'' field (`ndb_sort_by_server(cdb, NDB_SORT_NUM, ''number of call'')`) allows a logical ordering of the calls database. Initially, the call numbers *are* contiguous, however, through deletions, gaps may be left in the call sequence. The renumbering of the calls upon the occurance of a deletion would be too impractical because all sound segments use these numbers in their filenames. Renaming each sound segment would be monumentously slow. Note that the call number is only unique throughout the call's lifetime. Once a call is *completely* deleted (i.e. *erased from disk* as opposed to *marked deleted*) the call number may be used again.

The field ''type of call'' may have one of three values: ''Known caller message'', ''Known caller hangup'' or ''Unknown caller''. An unknown caller is a person who did not identify himself/herself to *Phone Slave* via the telephone keypad login technique. It is obvious, therefore, that the field ''caller name'' and ''caller phone id'' are not expected to be present in that record. A known caller may or may not chose to leave a message, hence the differentiation. The field ''segment times'' must be present in any call that results in a message. The field value is a list of blank-space separated message segment times in milliseconds. An unknown caller is expected to leave five message segments--one for each question asked by *Phone Slave*--while a known caller may only leave one. Any discrepancy is noted by `verify_sound_files()`, a function that matches up the sounds recorded on disk with the sounds expected by the calls (and users)

databases. The message segments are stored under the filename dictated by "sound filename prefix". Each segment is stored separately by appending the segment number to the sound filename (e.g. "...\call1_0, ...\call1_1 through ...\call1_4 for the five segments of call one).

The date and time of the call are stored in two fields. In "call date, UNIX time", the time is recorded in some esoteric unix format. This field is neither used by *Phone Slave* or *XMess*. It is supplied, however, in the event future versions may need the information. The field "date of call" records the information in an English readable format. Again, *Phone Slave* does not use this information (though perhaps it should), but *XMess* does make the information available to the user.

Finally, "status flags" contain the present state of the call: new, unread, or deleted. Deleted calls, marked with a 'D' in the first position in the field, are calls that the user wishes to remove but have not actually been deleted from the database or the disk. Deleted calls do not appear on the *XMess* display, and it only becomes a matter of time before they are completely erased with the function `expunge_calls()`. New and unread calls are marked with an 'N' or 'U' in the second position of the status field. New calls are those that *Phone Slave* and *XMess* have not notified the user about. Unread calls are those that the user knows exists (e.g. when *Phone Slave* announces "You have six new messages") but has not yet read. The supporting functions for reading and setting these fields are located in "flaghacks.c".

5.4 Users_db format

The users_db maintains a list of the known users and their personal messages. This database is also shared by the rolodex application *XRolo*. All of the records and the fields in users database, therefore, are not meaningful to *XMess* and *Phone Slave*. The extra

information in the database not needed is simply ignored. The following is an example of a users_db (without any of the *XRolo* extraneous information that might normally exist):

```
Start of DB {
Record {
{name} {Lorne Berman\00}
{username} {lberman\00}
{pending messages} {1\00}
{name sound} {Master Lorne\00}
{telephone id} {7635\00}
}

Record {
{name} {Chris Schmandt\00}
{username} {geek\00}
{telephone id} {2021\00}
{last message left} {1\00}
{sound filename prefix} {\\lberman\\snd\\2021.\00}
{segment times} {12548 \00}
}
}
```

The users_db must have one entry corresponding to the owner. In this example, the owner's record is the first entry. Both phone applications identify the owner record by the user's username. Upon the startup of either *XMess* or *Phone Slave*, `get_users_db()` finds the record that contains the username of the person who started the program. If none exists, the application exits.

The full name of the owner (as well as any known user) may be found in the field "name". The existence of this field, however, is not essential. Likewise, whenever the dectalk voice synthesizer pronounces a name, it will use "name sound" if the field exists.

The owner's list of new messages is stored in "pending messages", but *XMess*

makes no use of this field. Similarly, "last message left" contains the call number of the user's last message, but XMess also ignores this field. *Phone Slave* uses it to inform a known caller that his/her message has been read by the owner.

XMess identifies the record of a known user through the presence of "telephone id". If this field does not exist (which may be true in many cases), then the record is ignored until the field is added. This may be accomplished through the use of the known users' popup, or one may simply edit the database manually.

If the owner leaves any personal messages to a known user, the filename header is stored in "sound filename prefix" and the message lengths are placed in "sound segments". Both these fields have the same function in users_db as they do in calls_db. The number of messages and their lengths may be extracted from the field "sound segments", while their filenames may be constructed by appending the message number to the filename prefix. The filename prefix, instead of conveying information about the number of the call (e.g. "...\call1_0") contains the telephone ID of the user that will receive the message. For example, a single personal message to Chris Schmandt would have the filename ".../2021.0".

5.5 Motd_db format

The motd_db maintains the list of all possible outgoing messages and the number of the current outgoing message. This database has only one record, as shown below.

```
Start of DB {
Record {
{current generic message} {2\00}
{0 length} {2296\00}
{1 length} {2828\00}
{2 length} {3420\00}
{3 length} {4564\00}
{0 name} {snd\\fld_motd.snd\00}
{1 name} {snd\\motd.snd\00}
```

```
{2 name} {snd\\motd2.snd\00}
{3 name} {snd\\motd3.snd\00}
{0 title} {Recorded\00}
{1 title} {Generic 1\00}
{2 title} {Generic 2\00}
{3 title} {Out to Lunch\00}
}
}
```

Three fields are associated with each message: "name", "length", and "title". The "name" field is the actual filename under which the message is stored. The length contains the duration of the message in milliseconds. Finally, the title is used to "pretty print" the name of the message. In the outgoing message popup, the title, if available, is used to reference the message. This field exists because it is much more meaningful for a user to encounter a button labeled "Out to Lunch" than a button that read "motd3".

On startup, if the function `get_motd_list()` does not find the motd_db already loaded or on disk, a default outgoing message must be created. The default is an empty sound file which *Phone Slave* may use to store remotely recorded messages.

To gain access to the motd_db, `get_motd_list()` may use one of three distinct methods: grabbing, loading or creating. (This is in contrast to the two ways in which a calls_db or users_db may be accessed, e.g. grabbing or loading.) The function may grab the motd_db handle from the common record, if it exists. If not, the database may be retrieved from disk. Finally, if both fail, the function creates an empty database which is then filled with the backup-list information. Some functions *always* expect to find the backup-list in the motd database, therefore, it would be incorrect to create or use a database which violated this requirement.

Similar to the message segments in the calls_db and users_db, the sounds in the motd_db have associated sound lengths. Although *XMess* requires this information for

the sound viewer widgets, *Phone Slave* has no such need. The motd sounds, therefore, are checked in `get_motd_list()` for the presence of their lengths with the function `validate_motd_length()`. This function returns the length of a sound from the information in the database, or if this is not available, it queries the sound server for the sound length, places the datum in the motd_db and then returns the length.

The motd_db, similar to the calls_db and users_db, has a self-cleaning mechansim. Any sounds found in the motd directory that are not present in the motd_db are deleted from the sound server. This task is performed by the function `verify_motd_db()` which may be called at startup or exit to avoid any delays that usually accompany such deletions.

6. Walking Through XMess

The purpose of this chapter is to step through *XMess* operation from start to finish. Hopefully, this will reveal to the programmer how the individual pieces (described in the previous chapters) interact to form the whole network of *XMess* operations.

6.1 Startup Procedures

As described in chapter two, *XMess* sets up the basic window display--e.g. the icons are loaded, the popups are created and the translations are modified. These window operations, which occur in `main()`, only form the "frame" of *XMess* windows. Still lacking are call, known user, and outgoing message widgets. The creation of these widgets is segmented into distinct procedures which may be called upon anytime during program execution. This allows *XMess* to handle most any database modifications that *Phone Slave* may perform while its running.

XMess then calls `inet_open()` in an attempt to contact *Phone Slave*. In that procedure, the global variable `inet_conn` is set according to the state of *Phone Slave*. This information is used in four instances: communicating with *Phone Slave*, communicating with the sound server, determining when *XMess* should delete files, and widget display. At this time, it is the widget display which is affected by `inet_conn`. The function `fix_slave_wids()` is called, which "sensitizes" (i.e. allows the selection of) the "slave:on/off" widget and the "speaker:on/off" widget. Both these buttons are enabled if *Phone Slave* is connected, otherwise they are disabled.

At this time, the calls and users databases are opened with `open_calls_db()`. This function first sets up the connection with the *netdb* server, and then uses `get_calls_db()` and `get_users_db()` (as described in chapter five) to load or grab the handle to the databases.

6.1.1 Loading Calls

Any calls pending in in the calls_db are ready to be displayed. The procedure `load_calls_db()` serves this purpose, and is called at this time. This function examines each call record, and any calls that are *not* displayed or deleted (i.e. have its "deleted" status flag set) have a call widget created with `create_but_and_sv_from_db_rec()`. If any calls exist in the *XMess* domain but are marked deleted, the widgets for the call are destroyed with `kill_but()`. Non-deleted calls which are found in both calls_db and *XMess* are left undisturbed. After the examination of the calls databse, if any new calls have been created, *XMess* begins flashing its icon while the new calls are displayed with `manage_all()`. Because of the non-destructive nature of `load_calls_db()`, this function may be called at any time during *XMess* execution to update the display *without* disturbing normal operation.

The purpose of `create_but_and_sv_from_db_rec()` is to examine a call record and produce the proper parameters needed for `create_but_and_sv()`. First, the function determines the type of the call, either known or unknown user. This allows the button label field to be set. If the call type is a known user with no message, the function only has to set the date field and jump to `create_but_and_sv()`. Otherwise, the message segment field must be parsed. From each segment, the sound name and sound length parameters are built. If the number of segments is at a maximum, then the standard placement of the messages (one segment under each title) is used. The only exception is when a segment seems particularly short (length < `MIN_SOUND_LEN`), in which case the segment is not displayed and appropriate field is left blank. When *XMess* encounters a message with only one segment (e.g. a known user who left a message) the sound is placed in the last column. This function returns after it positions the newly-created call widget at the bottom of the display queue with

`place_but_last()`.

6.1.2 Connecting to the Sound Server

The sound server is the device (presently an IBM XT with a Dialogic sound board) that allows *XMess* to play and to record digitized sound. Unfortunately, the present sound server is only capable of performing one task at a time, with no provisions for queueing multiple requests. This creates a problem when two processes, such as *XMess* and *Phone Slave*, need to share the same sound server. In this situation, the two processes must negotiate for control of the server. The negotiation is accomplished via the socket connection. Whenever *XMess* wishes to access the sound server, the socket command "Stop waiting for ring" is sent to *Phone Slave*. At its convenience *Phone Slave* responds with an acknowledgment. In this mode, *Phone Slave* is "off". It can not answer any incoming calls, nor may it request for the control of the sound server. It must wait until *XMess* passes the control back with the socket command "Start waiting for ring".

In the function `get_pc()`, which is called after the loading of the call widgets, *XMess* opens a connection to the sound server. Before this is possible, *XMess* must be sure that *Phone Slave* (if running) is not engaged in answering a call. *XMess*, therefore, waits for the arrival of either "Phone Slave is idle" or "Phone Slave is active"--one of which *Phone Slave* sends upon the opening of the socket connection. If *XMess* receives the former, *Phone Slave* is off and *XMess* may proceed. Otherwise, if *Phone Slave* is on, it must be turned off before *XMess* accesses the sound server. This is done immediately if *Phone Slave* is **not** currently answering a call, or delayed if *Phone Slave* is recording a message--which is the situation if *XMess* receives "Incoming call being taken".

While *Phone Slave* is off, `get_pc()` performs a variety of tasks which involve

communicating with the sound server. First, a handle to the server is obtained. If successful, *XMess* "greets" the server by asking for its name. The server directory (which should contain the subdirectories `calls`, `snd` and `sounds` for the incoming calls, outgoing messages and greeting sounds, respectively) is then set to the owner's username. Finally, the functions `get_motd_list()` and `verify_motd_db()` are called to load and verify the outgoing calls. These procedures are located here, instead of near the calls_db and users_db loading functions, because of the possibility that they may need to access the sound server--`get_motd_list()` calls `verify_motd_length()` and `verify_motd_db()` uses `s_rm()` (remove sound file). Before returning, `get_pc()` returns *Phone Slave* to its prior state (on or off) at the beginning of the function.

Upon return, *XMess* calls `display_motd_svs()` for the final display setup, and performs some basic housekeeping tasks. These tasks include initializing variables, realizing widgets, and setting the resource `XtNinput` to `True` for all of the shells.

6.2 Main Loop Events

After *XMess* finally performs all the necessary startup routines, it sits and waits in `XtMainLoop()`, because, as with all properly programmed X applications, *XMess* is event driven. In this case, there are three distinct classes of events that *XMess* recognizes:

1. Normal window operations
2. Timer expirations
3. Signal IO

Window events (such as a button press) are handled with standard callback and translation table mechanisms. Timers are also triggered with a special type of callback.

Strictly speaking, the IO signal handled by `inet_datain()` is **not** an event. Such a signal is transparent to the X server, and the event dispatching routines in `XtMainLoop()` have no knowledge of its existence. Nonetheless, regardless of its origin, a `SIGIO` is handled in much the same way as an X event. The remainder of this chapter is devoted to explaining the various functions *XMess* performs when handling these events.

6.2.1 Phone Slave Command Actions

When *Phone Slave* sends a command, it is eventually parsed by *XMess* in `inet_parse_command()`. The recognized commands are listed and explained below:

"Die, please" -- *Phone Slave* is exiting. *XMess* closes the connection and calls `fix_slave_wids()` to notify the user of this development. If *Phone Slave* originally initiated the connection, *XMess* needs to call `inet_open()` to set up the socket (in the event *Phone Slave* comes back on line). Otherwise, the socket is in proper order and connection attempts are routed through `inet_connin()`.

"Phone Slave is active" -- *Phone Slave* is on and waiting for a call. This command may be sent by *Phone Slave* when either of the applications are booted to notify *XMess* of its state. The widget "Slave" is changed to read "Slave:ON".

"Phone Slave is idle" -- *Phone Slave* is off and in an idle state. If appropriate, this command is also sent when one of the programs is started.

"Incoming call being taken" -- *Phone Slave* is about to answer the phone. *XMess* sets the variable `slave_busy` to `True` and then calls `inet_says_in_call()` which pops up the incoming call window. While `slave_busy` is `True`, *XMess* does not attempt to gain control of the server. This act would disconnect the call.

"Known user has been identified" -- The current caller has identified herself with the telephone keypad login technique. *XMess* strips off the caller's telephone ID which is appended to the front of the command string. With this information, the users_db is scanned for the caller's name, which when found, is printed in the incoming call window. The "XRolo" button is then sensitized to allow the display of the caller's rolodex card.

"Incoming call has ended" -- *Phone Slave* is finished with the call. *XMess* returns the variable `slave_busy` to the `False` state and then calls `inet_says_end_call()` which pops down the incoming call window.

"Calls db has been modified" -- *Phone Slave* sends this command after each incoming call. *XMess* simply performs a `load_calls_db()` to update any changes that have been made.

"Motd db has been modified" -- This is received by *XMess* after *Phone Slave* modifies the motd_db. Since the only motd function *Phone Slave* performs is the recording of a new outgoing generic message, the only change in the database would be in the length of motd #0. *XMess* updates the outgoing message window by calling `display_motd_svs()`.

"Command Acknowledged" -- After performing a request sent by *XMess*, *Phone Slave* returns this command. Only when *XMess* receives the acknowledgement does it perform any actions which pertained to the command sent. This is accomplished through the use of `inet_last_command`. This variable is examined so that *XMess* knows the source of the acknowledgment, and then the appropriate functions are called.

6.2.2 XMess Command Actions

The following is a list of commands, their functions, and the process in which they are invoked by the user:

"Monitor Calls" -- This command tells *Phone Slave* to turn on the speaker-phone. It is invoked when the user attempts to toggle the "Speaker:OFF" widget which is located on both the main display and incoming call windows. On acknowledgement, *XMess* changes the button to read "Speaker:ON".

"Stop monitoring calls" -- Similar to "Monitor calls", this command instead tells *Phone Slave* to turn the speaker-phone off.

"Motd db has been modified" -- This command notifies *Phone Slave* that the motd_db has been altered by *XMess*. After the user records or deletes of an outgoing message, the command is sent. The motd_db is the only database for which *Phone Slave* caches any information. *XMess*, therefore, never sends "Calls db has been modified" because *Phone Slave* extracts call data directly from calls_db. Similarly, no command exists to notify either application that the users_db has been changed because neither program caches any known user information.

"Hang up the phone" -- When *Phone Slave* is taking a message and receives this command, it immediately stop recording and halts its dectalk output. The phone, however, is actually kept off-hook to allow the user to pick up the receiver (or speaker-phone) and converse with the caller. The command name, therefore, is somewhat of a misnomer. This action is performed when the user clicks on the "Answer" button inside the incoming call window.

"Stop waiting for ring" -- This command effectively turns *Phone Slave* off. In this mode, *Phone Slave* sits idle and ignores any incoming calls. There are two situations

where *XMess* will send this command. First, the user may manually toggle *Phone Slave* off with the command button "Slave:ON". Upon the acknowledgment that this has been done, *XMess* (or rather `inet_parse_command()`) changes the button label to "Slave:OFF". Second, if *XMess* needs to access the sound server, it must first send this command to avoid possible conflict with *Phone Slave*.

"Start waiting for ring" -- *Phone Slave* is turned (back) on with this command. The phone, if off-hook, is placed back on-hook and any call is disconnected. Any detected incoming calls are answered. There are three situations where this command will be sent. First, through the use of the "Slave:OFF" button. Second, after the user "answer"s the phone, eventually he will finish the call with "hangup". The "hangup" callback uses this command to place the phone on-hook. Third, if *XMess* halted *Phone Slave* because it needed to access the sound server, this command may be sent to turn *Phone Slave* back on. This is accomplished through the use of the X timer mechanism. When the sound server finishes playing a sound, a timer is set. Upon its expiration--assuming the sound server has not been accessed in that time--*Phone Slave* would be set to its initial state (i.e. the state it was in *before XMess* used the sound server).

6.2.3 Playing a SoundViewer

The SoundViewer is the most prolific of widgets in all of *XMess*. Three of the major windows use them. The functions that control the SVs, therefore, must be general enough to accomodate all of their different enviornments.

In all cases, before a SoundViewer begins to play, *XMess* must be sure that it has exclusive access to the sound server. In all the callback routines that start a SoundViewer, the status of *Phone Slave* is examined with `check_slave()`. If the *Phone Slave* is idle (i.e. "Slave:OFF"), the function continues as normal. Otherwise,

Phone Slave must be turned off by sending "Stop waiting for ring". *XMess* could sit idly in a loop, continuously checking the return value of `check_slave()` and proceeding with the function when *Phone Slave* has been turned off. This method is undesirable because it wastes processor time and does not allow the dispatch of any interim X events. *XMess* instead returns from the function after setting the variables `where_comm_go`--which contains an integer represention of the calling function--and `where_comm_wid`--which is the handle to the SoundViewer that will be played. When the acknowledgment is received, `inet_parse_command()` examines `where_comm_go` and jumps to the appropriate routine. This method allows *XMess* to execute `XtMainLoop()` while waiting for *Phone Slave* to be turned off, which is most acceptable.

7. References

1. Christopher Schmandt and Barry Arons. Phone Slave: A Graphical Telecommunications Interface. Digest of Technical Papers, IEEE International Conference of Consumer Electronics, 1984.

2. Christopher Schmandt and Stephen Casner. Phonetool: Integrating Telephones and Workstations. Proceedings of GlobeCom '89, IEEE Communications Society Conference, November 1989.

3. David Anderson. Phone Slave II: A Modular Portable Reimplementation. Bachelor's Thesis, MIT, September 1989.

4. Christopher Schmandt and Barry Arons. Desktop Audio. Unix Review, October 1989.