PhoneStation, Moving the Telephone onto the Virtual Desktop

Stephen A. Uhler

Computer Science Research Division
Bellcore

1. Introduction

For over a decade now, the workstation has been viewed as an electronic desktop, with multiple windows on the computer screen as the metaphor for a desk.[1] This electronic desktop has become the focus for dealing with office information. The telephone, although an important component of an actual desktop, has not yet been integrated into the modern desktop environment.

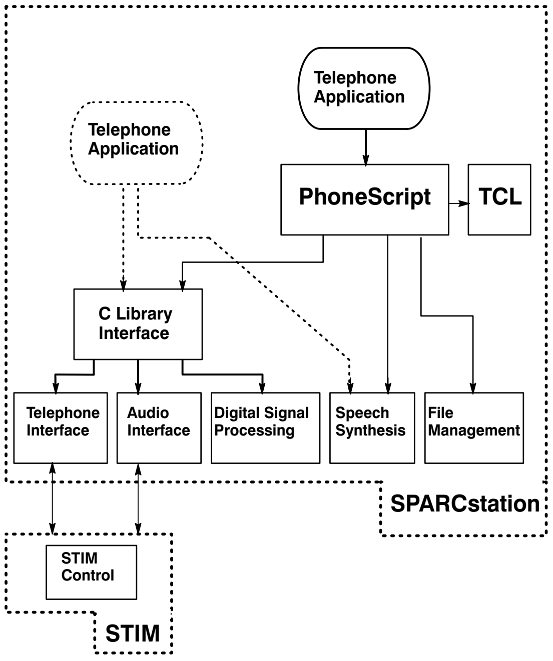
It should be possible to receive audio telephone messages as ordinary electronic mail (**e**mail), thereby taking advantage of the many message management capabilities we have become accustomed to in **e**mail systems. A unified interface to handle voice mail and **e**mail would eliminate the distinct and increasingly more complex user interfaces to telephone answering machines or voice mail systems, and provide the ready exchange of information between the computer and the telephone.

With the telephone an integral part of the computer desktop, many new applications come to mind. While retrieving voice mail messages over the telephone why not have the answering machine application convert your regular **e**mail to speech, and read it to you as well. If there is a fax machine available, as is the usual case at a hotel or conference, you could instruct the answering machine application to have the workstation fax you your regular **e**mail. Once on the phone, connected to your workstation, why not fax that article you forgot to bring with you, or that viewgraph you didn't think you'd need.

This paper will describe the components of **P**hone**S**tation, a system that provides a Sun SPARCstation with complete control over an ordinary telephone line. After briefly describing the **P**hone**S**tation hardware and basic software facilities, it will describe in detail, **P**hone**S**cript, the **P**hone**S**tation high level language for building interactive telephone applications.

2. PhoneStation System Components

Figure 1: PhoneStation Architecture



Phone**S**tation runs on a Sun Microsystems SPARCstation. The system consists of some hardware "glue" that enables the SPARCstation to interface to a telephone line, a suite of software support libraries, and **P**hone**S**cript, a language for building

and **P**hone**S**cript, a high level procedural language that uses phone based applications.

ABSTRACT

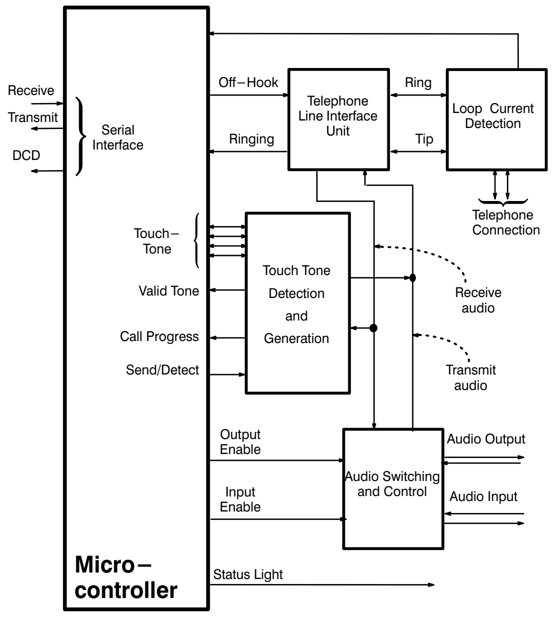
Phone**S**tation is a system that provides a Sun Microsystems SPARCstation† with complete control over an ordinary telephone line. It consists of a telephone line interface unit with loop control and touch tone detection, a suite of supporting software libraries that include digital signal processing for call progress monitoring, text-to-speech conversion, telephone line control,

interactive telephone applications. The software components of **P**hone**S**tation are shown as boxes in Figure 1. The basic support libraries are along the bottom. Applications programs are normally written in the **P**hone**S**cript language, although they can be written in C, and call the underlying library routines directly.

2.1. PhoneStation Hardware

The SPARCstation hardware interface, called STIM (SPARCstation Telephone Interface Module), connects to the SPARCstation through a serial port and the audio connector. It is assembled from off-the-shelf components, and fits in a 2" x 4" x 4" box. A block diagram of the STIM hardware is shown in Figure 2.

Figure 2: STIM Block Diagram



The core of the STIM is the single chip computer, a Zilog Z8[2]. The Z8 has 16 individually controllable I/O (input/output) lines, three of which are configured as an rs232 serial port. The remaining I/O lines are connected to a telephone line interface hybrid, a touch-tone detection and generation chip, a telephone loop current detection relay, and a pair of audio switching relays. The telephone line interface unit provides the required isolation from the telephone line. In addition to inserting and extracting audio signals from the telephone line, it detects ringing, and can place the telephone line in either the on-hook or off-hook state. The touch-tone detection chip does just that; detect the presents of touch-tones, which are converted to ASCII signals by the Z8, and sent over the serial interface to the SPARCstation. The loop sense relay monitors the state of the telephone line to detect when a telephone call has terminated. The audio switching relays permit other audio devices to be connected to the SPARCstation when the STIM is not in use.

The program that runs in the Z8, written in *basic*, communicates with a process on the SPARCstation using single letter ASCII commands via the serial port. The digital-to-analog and analog-to-digital conversion capabilities of the SPARCstation are used to play and record digitized audio.

2.2. PhoneStation Software

The primary application interface to **P**hone**S**tation is **P**hone**S**cript, a command interpreter that uses TCL (Tool Command Language)[3]. TCL, written by John Ousterhout of Berkeley, is a freely available library of C routines that provide a software system with an embeddable shell-like command interpreter. This command interpreter is combined with a suite of software support libraries, written in C, providing digital signal processing (DSP) for call progress monitoring, a text to speech synthesizer using the ORATOR® speech synthesizer [4], and some relational file management routines that facilitate the storage and retrieval of data that may be required by a telephone application.

Phone**S**tation consists of 5000 lines of C code in the support libraries, and another 3000 lines of C code to interface them with TCL. There is another 250 lines of Basic that runs on the Z8 microprocessor in the STIM, as well as 1500 lines of C code that provides the development environment for programming the Z8 and configuring the DSP code.

The telephone interface module provides a device independent abstraction for interacting with the telephone line. It interacts with the STIM over a serial line. The software configures the SPARCstation serial line same way as a modem that is set up to permit both incoming and outgoing calls. An application that is waiting for incoming telephone call blocks (in open()) until a call arrives. When the STIM detects ringing on the telephone line, the Data Carrier Detect line of the serial interface is asserted, causing the open() to complete, and the application to continue. Additional information about the state of the telephone line is then passed between the STIM and the telephone interface module over the serial interface. Applications wishing to place outgoing calls can do so any time the telephone interface is not already being used, even if another application is waiting for an incoming call.

The audio interface module controls access to the SPARCstation audio device. It set the play and record volume levels, and controls the amount of audio data in the audio device driver queues. **P**hone**S**tation plays audio files by periodically sending batches of audio to the device queue. Application programs can change the batching interval to obtain more time for other computations before the next batch of audio is required.

The DSP module uses second order recursive digital band pass filters and energy detectors[5], running in software on the SPARCstation, to process the incoming audio stream and determine the status of a telephone call. Signaling tones used for telephony are simple combinations of pure tones (sine waves). The band pass filters isolate the sine waves, then the energy detectors determine if a signal is present at the required frequency. The DSP module identifies dial tone, ringing, and busy signals, which are used to monitor the progress of an outgoing call. Modem tones and voice patterns are recognized once the call has been completed. Routines are available to detect and decode touch-tones as well, even though in the current version of **P**hone**S**tation, the touch-tone detection can also be done by the STIM in hardware. The signaling tones, as well as various answering-machine like "beeps," are synthesized by the digital signal processing module as needed.

The text-to-speech synthesizer runs as a background process, and has been optimized to pronounce names and addresses accurately, although it can synthesize arbitrary text quite well with a little coaching. The synthesizer typically takes less time to synthesize an utterance than it takes to speak it. Synthesized output can either be sent to the telephone directly, or saved in a file for later use.

The file management module provides a simple relational abstraction of a file that integrates structured file access into the **P**hone**S**cript language. It provides access to the files contents through TCL variables, and supports the selection of items in the files through the evaluation of TCL expressions containing references to specific items.

3. The PhoneScript Language

Telephone applications are similar to many real time process control applications. They have real time constraints; the phone must be "answered" within a certain time, or a touch-tone received from the user must be processed before the next one arrives. Time out conditions abound: how many rings to wait before "answering" the telephone, how long to wait for a dial tone, and how much time to listen for a touch-tone, are examples of just a few. Most of the inputs into the system come in the form of asynchronous events, they can occur at any time, and often do.

A typical method for dealing with this type of system in a language such as C, is to use an event driven state machine. The program waits for an event, acts upon it, transitions to the next state, and waits for the next event to occur. Although state machines can often be implemented efficiently, they get complicated quickly, as even a simple application can have many states. In those cases where several things are happening at the same time, such as playing instructions to the user while listening for touch-tones, the complexity is compounded. The complex code required to manage all of the events, timeouts, and exceptions often obscures the primary intent of the application code.

3.1. PhoneScript Language Design

Phone**S**cript was created to provide a programming environment that makes writing interactive telephone applications easy to do. To achieve this end, the **P**hone**S**cript language was designed with several goals in mind. Simple applications should be short, and easy to write. More sophisticated applications should be possible, with the basic structure of their simpler cousins retained. Adding just one more feature to an application should not require a complete re-write of the code, just a minor addition. When the application is completed, the basic structure of the code should match its conceptual structure. One shouldn't have to be a contortionist to translate the application into the language. **P**hone**S**cript is a language intended for interactive applications. Each complete interaction, or "transaction" with the user, should be captured by a single language construct. The design of interactive applications is hard to get right the first time. Consequently its important that applications are easy to debug and modify, with an incremental style of application refinement encouraged. Finally, it should be easy to interface telephone applications to existing systems and applications, such as Fax, electronic mail, or graphical user interfaces.

Phone**S**cript consists of the 13 telephone interface commands listed in Table 1.

Table 1 PhoneScript Summary of Commands

Name	Command Description
audio	Low level control of the audio system
beep	Play *beeping* tones
call	Place an outgoing phone call
cnv2tt	Convert an alphanumeric string to touch-tones
db	Structured file management commands
debug	Interactive debugging
hangup	Hangup the telephone line
on	event processing
phone	Direct phone line interface manipulation
play	Play audio files and receive touch-tones
prdate	Date and time conversion and formatting
record	Record an audio file
synth	Text to speech conversion

These commands are used in conjunction with the built-in functions of TCL. I will not fully describe the TCL language here. Instead, I will note only the features of TCL required to follow the **P**hone**S**cript example programs. TCL provides the typical expression evaluation primitives, flow control constructs (such as *for, while, if-then-else* and *switch),* and procedures typically found in procedural languages. TCL operates on white space separated lists of ASCII character strings that are terminated by new lines or semi-colons (;). The first string in a list is the command, with the remaining strings passed to the command as arguments. White space may be included in a string enclosing it in quotes ("), or by surrounding the string with braces ({}). The use of braces, which may be nested, also prevents variable and command substitution. Brackets ([]) are used for command substitution where *[command]* in TCL is analogous to *‘command‘* in the shell. The value of a variable is obtained by *$variable,* or *$variable(member)* for an array, where a backslash (\) can be used to prevent the special meaning of *$*. TCL also provides a wealth of built-in string and list manipulation commands. The **P**hone**S**cript functions in Table 1 are added to the core TCL commands to provide the telephone application specific capabilities of **P**hone**S**cript.

Phone**S**cript uses the notions of event handling and implicit iteration to provide a framework for straight forward application development. Since **P**hone**S**cript is intended primarily for interactive telephone based applications, all of the setup and initialization of the telephone, audio, and DSP sub-systems is taken care of automatically, with many configurable parameters set to useful default values.

As an interpreted language, **P**hone**S**tation simplifies program development by allowing interactive debugging of applications. The low level time critical tasks are handled within compiled C code, so actions that happen at the interpreter level are *human response* kinds of actions. Several tenths of a second response time for their execution is not objectionable.

The **P**hone**S**cript main program manages most of the required book keeping. It initializes the telephone line interface, and the audio and text-to-speech sub-systems. Application programs use special global variables to customize the initialization of the system. The semantics of TCL are extended to permit command arguments of the form: *keyword=value*. When included as a command argument, they override the global value of the *keyword* parameter for the duration of the command. Applications can set useful default values at the top of the program, then override them on a command by command basis.

3.2. Sample PhoneScript Applications

The following PhoneScript examples, which are complete, working **P**hone**S**cript programs, will be used to illustrate the key features of the **P**hone**S**cript language. In the examples, items printed in `this font` represent **P**hone**S**cript code fragments or commands.

Figure 3: Phone**S**cript Version of **Hello World**

```
#!/usr/local/bin/PhoneScript
# place a call say: hello world

set usage "Usage: $argv(0) <number>"
if {$argc < 2} {
  puts stderr $usage; exit 0 }
synth "Hello World."
call $argv(1)
play # until .
exit 0
```

† SPARCstation is a trademark of Sun Microsystems

The first example is shown in Figure 3. This is the **P**hone**S**cript version of the *Hello World* program. The **P**hone**S**cript version synthesizes the phrase *Hello world*, places a phone call to the number specified on the command line, and speaks *hello world* when the called party answers the telephone. **P**hone**S**cript imports the command line arguments and the environment from the shell, so **P**hone**S**cript programs can be run directly from the shell. The `synth` command controls the text to speech synthesizer. It works in the background, leaving the resultant audio data on a queue when the synthesis is complete. The `call` command places the telephone call, and `play` sends the synthesized audio to

the
telephone line. The '#' instructs `play` to use the `synth` queue, instead of looking for a pre-recorded audio file. Although this example does quite a bit more than the standard C language version of *hello world*, it requires about the same amount of code. The setup required to operate the telephone line is handled automatically.

The second example, shown in Figure 4, is a simple, yet functional answering machine application. When the telephone rings, **P**hone**S**cript waits for 3 rings (the default), answers the telephone, then plays a pre-recorded greeting message. After the *beep* the caller can leave a message, which is saved as digitized audio in a file, and forwarded via electronic mail to the **P**hone**S**cript user. The remaining examples will build upon this one to enhance its functionality and to explore features of the **P**hone**S**cript language.

Figure 4: Complete Answering Machine Program

```
set greeting $HOME/message.au
set action voice2mail
set timelimit 0

on call {
   set msg msg.[prdate].au
   exec touch $msg
   play $greeting until #
   beep
   record  $msg "" record_timeout=450
   }

on hangup {
   exec < $msg $action
   }
```

† SPARCstation is a trademark of Sun Microsystems The variables `greeting`, `action`, and `timelimit` are initialized when the application begins. The `greeting` variable contains the outgoing greeting message, which can be recorded either by using another **P**hone**S**cript application, or with any of the standard audio applications that are available on the SPARCstation, such as *soundtool*[6]. The variable `action` contains the name of the UNIX command that will be invoked to deal with the message left by the caller. The digitized audio representation of the message is available as the standard input to that command. The `timelimit` variable is one of many **P**hone**S**cript configuration parameters. It sets the time to wait for the user to reply to
UNIX†

[5] † UNIX is a trademark of Unix Systems Laboratories.

a greeting message before proceeding to the next command. In this example, we with to start recording a voice message as soon as the greeting is finished playing, so the `timelimit` is set to zero.

Unlike the *Hello World* example, where each statement is executed sequentially, the bulk of the work in the answering machine is done by the event handling constructs, `on call` and `on hangup.` **P**hone**S**cript waits for a phone call to come in, answers the telephone, then runs the body of the `on call` command. The **P**hone**S**cript `prdate` command returns the UNIX time, which is used to name the message file. The `play` command plays the prerecorded audio message. By default, `play` plays the audio message to completion. The `until` keyword specifies a regular expression that causes `play` to terminate immediately if the touch-tones keyed by the user match the expression. In this case, keying the '#' key on the telephone keypad will cause the answering machine program to skip over the rest of the greeting, *beep*, then start recording. After the `on call` code is concluded, either because the caller hung up, or the message time limit was exceeded, **P**hone**S**cript hangs up the telephone, then runs the body of the `on hangup` command. The TCL built-in command `Exec,` calls `voice2mail`, a short shell script that converts the digitized voice message into a MIME format multi-media **e**mail message[7] by encoding it in ASCII, prepending the appropriate mail header lines, and forwarding it on to sendmail[8] for delivery. After the `on hangup` commands are finished, **P**hone**S**cript waits for the next call to arrive.

In **P**hone**S**cript applications that answer telephone calls, all but one time initialization code is in the body of one of the `on` event conditions, which are summarized in Table 2. The code associated with each event is read and saved during the initial scan of the **P**hone**S**cript program, but it is parsed and executed only when the corresponding condition occurs. This event handling mechanism in **P**hone**S**cript allows applications to deal with the asynchronous nature of the application domain in a straight forward manner.

Table 2.
Summary of PhoneScript Event Conditions

Name	Event Description
on call	A telephone call is answered
on endringing	The telephone stopped ringing before the call was answered
on hangup	The telephone call was terminated
on int	The PhoneScript program was interrupted from the keyboard

on ring	The telephone started to ring.
on signal	The **P**hone**S**cript program was signaled by another process

One of the primary benefits of **P**hone**S**tation is its ability to use the telephone as simply another user interface to the workings of the computer. If the answering machine program is running all of the time, there needs to be a mechanism for escaping from the answering machine into more sophisticated telephone based applications. One way to accomplish this is to have the user key in as touch-tones a secret code while the answering machine is playing its greeting, a common technique used in consumer answering machines. In **P**hone**S**cript we can create any number of applications, and assign each one its own sequence of touch-tone codes. The name of each application will be the code needed to invoke it.

To accomplish this, the commands in Figure 4 are replaced by the code in Figure 5. The lines that have been emboldened mark the changes.

Figure 5: Revised Play Command

```
set greeting $HOME/message.au
set action voice2mail
set timelimit 0

on call {
  set msg msg.[prdate].au
  exec touch $msg
  play $greeting until # unless {
    if {$unless(tone) == "*"} hangup
    continue
    }
  catch {source $tones.tcl}
  beep
  record  $msg "" record_timeout=450
  }

on hangup {
  exec < $msg $action
  set tod [prdate "%A, %l %M %p."]
  synth $tod to $msg.tod
  }
```

† SPARCstation is a trademark of Sun Microsystems

Until now, `play` has been used to play an audio file and (optionally) stop after receiving a touch-tone. In the general case, a single `play` command can be used to support an entire transaction with the user, playing many audio files, and using touch-tones keyed by the user to guide the sequence in which the files are played. The `unless` option to `play` causes the TCL expression after the `unless` to be run any time a touch-tone is keyed by the user. While in the `unless` expression, a number of special **P**hone**S**cript variables that describe the current state of the `play` command are available, and can be examined or changed to customize the action of `play.` Using this technique, the special cases and exceptions can be handled from within a single `play` command, eliminating the need to bury a single user interface transaction in a maze of *ifs* and *elses* that would ordinarily be required to manage the special cases.

The TCL array `unless` contains a member for each of the variables passed by `play` to the `unless` expression. The just keyed touch-tone is stored in `unless(tone).` and the accumulation of touch-tones keyed in so far is stored in `unless(tones).` With this variation of the answering machine, when the user keys a `'*'` on the telephone keypad, the answering machine program executes a `hangup`, which immediately hangs-up the telephone line. This is invaluable in those cases where the answering machine picks up the call just as you are about to. When the greeting message is finished playing (or the user keyed a '#'), the variable `tones`, which is set by `play` as it finishes, contains the list of touch-tones entered while the `play` command was running. Normally `play` will stop playing voice files whenever a touch-tone is entered, as this is usually the desired behavior. In this application, the `continue` command instructs play to continue playing the message file even though a tone has been received. However, the '#' tone will still skip the rest of the greeting message, and proceed directly to the *beep* .

After the `play` command is finished, the TCL `source` command runs the application program (if any) whose name matches the tones entered. If the user keys the touch-tones *123#*, the answering machine program will attempt to include the program `123#.tcl`. The TCL `catch` command prevents the answering machine from flagging the error if the file *123#.tcl* does not exist.

Planning ahead for the next example, two additional commands are added to the `on hangup` expression, that will cause a *time of day* file to be created with each voice message. As before, `prdate` formats the current time and day, this time in a manner that can be easily read aloud. The argument to `prdate` calls the UNIX strftime() function, which replaces the *%X* constructs with the appropriate date and time strings. The `Synth` command converts the time and date string to speech, and saves it in a file. If the file is later played, it will say something like *Tuesday, eight fourty-six PM.*

This example demonstrates the **P**hone**S**cript notion of implicit iteration. The

behavior
of the `play` command is guided by user input. As new features are added to the interaction, the additional functionality is expressed from within `play`, with out having to restructure the code. With this added functionality, the answering machine application functions as a gateway to many other applications. New features are added to the answering machine by creating the functionality as a separate **P**hone**S**cript program fragment. The user accesses the function simply by entering its name.

The next example, in Figure 6, a voice message browser named *123#.tcl,* is accessed from within the answering machine by entering the touch tones *123#* while the greeting message is playing.

Figure 6: Program 123#.tcl - A Message Browser

```
set msgs [glob "{intro,msg.*,done}.au"]
set reason=tf; set timelimit=0

play $msgs until "9" unless {
  case $unless(tone) in {
    "#" {incr unless(file); beep}
    "0" {play $unless(file_name).tod}
    "1" {incr unless(file) -1; beep}
    "*" {incr unless(file_pos) -16000}
    ""  {beep}
      }
   continue
   }
hangup
```

† SPARCstation is a trademark of Sun Microsystems

This example shows how a single `play` command can be used to manage a complex transaction with the user.

First we use the TCL builtin `glob`, that works like the *csh* command of the same name to create a list of the current voice mail messages. The files *intro.au* and *done.au* are pre-recorded messages, that contain the audio equivalent of *Playing voice mail messages* and *Done playing messages* respectively. The `play` command plays the introductory message, followed by the voice mail messages, then the concluding message in sequence. Touch tones keyed in by the user are used to alter the playback sequence, as controlled by the `unless` expression.

The touch-tones '#', '0', '1', and '*' cause `play` to alter the default sequential playing of the messages. A '#' causes a skip to the next message, by incrementing the `play` variable *unless(file),* the current file in the message list. A '0' causes `play` to chime in with the time and date that the voice message was recorded, by playing the *time of day* file that was created when the message was recorded. A '1' causes the playback to skip backward to the previous message. Finally, pressing '*' causes the previous two seconds of the message to be re-played, providing another opportunity to write down the phone number you missed the first time. The little details, such as trying to skip backward before the first message, are dealt with automatically by **P**hone**S**cript.

Normally the `unless` expression runs only when a touch tone is entered by the user. However the configuration variable `reason` is set to alter the conditions that cause `unless` to run. In this example, when a message is finished playing, and the next one is about to start, the `unless` expression is run. The last case of the `case` statement, for which there is no touch tone, is taken when one of the audio files finishes, causing a *beep,* informing the user that the current message file has finished playing. Additional features of the message browser, such as deleting messages, or forwarding them to other programs, are easily added within this framework, by adding new cases into the `case` statement. The continue statement prevents `play` from terminating when the first tone is entered.

Once the voice mail message browsing is complete, it is unlikely that returning to the answering machine program to record a voice message is still desired. The `hangup` command causes the message browser to hang up the phone at once, skipping the message taking part of the answering machine.

Although this answering machine does the job, the `on ringing` event of **P**hone**S**cript, activated just as the telephone begins to ring, enables an application to made decisions about a telephone call before the answering the telephone. For example, if Calling Number Delivery[9] (sometimes called caller-id) is available, the TCL variable `number` contains the calling number when the `on ringing` section is run, so actions can be taken selectively based on the telephone number of the calling party. The code in Figure 7 is added to the answering machine program in Figure 5.

Figure 7: Select Actions Based on Calling Number

```
set caller_id 1
on ringing {
  if {[catch {source $number.tcl}]} {
    set greeting $HOME/message.au
    set action voice2mail
    set rings 3
    }
  }
```

† SPARCstation is a trademark of Sun Microsystems

The variable `caller_id` is set to turn on Calling

Number
Delivery, currently implemented by a readily available Calling Number Delivery interface, connected to the other serial port of the SPARCstation. When the telephone begins to ring, the `on ringing` code is executed before **P**hone**S**tation answers the call. As with the message browser in the previous example, if a file name matches the calling number, its contents are read and executed as part of the application. If no file exists, the greeting and action are reset to their default values. The variable `rings` is the count of rings to wait before **P**hone**S**cript answers the telephone. By creating a file whose name is the telephone number of the boss, a special message is played only when the boss calls. The contents of that file might contain:

```
 set rings 1
 set greeting boss.au
 set action "page_me 'The boss called'"
```

If it is the telephone number for the collection agency instead, the file might contain:

```
 set rings 99
```

even they don't have that much patience.

If Calling Name delivery is not available, the answering machine can still be programmed to choose different messages. This time the answering machine will be coupled with a configuration file to allow the greeting message and number of rings to wait before picking up the call to be chosen, based on the time of day and the day of the week. For example, the caller can be made to wait for 3 rings and be greeted with *good morning* on Tuesday mornings. If answering the telephone is not desired, **P**hone**S**tation can pick up the telephone at the first hint of ringing to play an appropriate message.

This feature is implemented in **P**hone**S**cript using a structured file. An example of which is shown in Figure 7.

Figure 7: Greeting Message Configuration File

```
days;start;end;greeting;message;no_rings;
0-6;630;1200;morning.au;;;
0-6;1200;1630;afternoon.au;;;
0-6;1630;1830;evening.au;;;
1-5;630;830;;;5;
1-5;830;1630;;work.au;2;
06;900;2100;;weekend.au;4;
0-6;0;2400;off_hours.au;default.au;1;
```

† SPARCstation is a trademark of Sun Microsystems

A structured file in **P**hone**S**cript consists of 1 or more line of text, each containing semi-colon terminated fields. The first line in the file names the fields, whose values are accessed via TCL variables of the same name. The remaining lines are the data. This configuration file has six fields. The first, *days* contains a range of days, 0 for Sunday, 1 for Monday etc. The next two fields give a range of times, in military time, for which this line applies. The fourth and fifth fields give the names of two pre-recorded message files that are played consecutively as the greeting message. The first message is used for a salutation, such as *good morning* and the other one for instructions, such as *Please leave a message at the beep.* The final field specifies the number of rings to wait before picking up the telephone.

This structured file is accessed through the **P**hone**S**cript `db` command, by including the code from Figure 8 into the answering machine program in Figure 5 instead of the Calling Name Delivery code.

Figure 8: Greeting Message Selection

```
on ringing {
  set day [prdate %w]
  set hour [prdate %k%M]
  set msg1 ""; set msg2 ""; set rings ""
  db select {[string match \[$days\] $day]}
  db select and "\$end > $hour"
  db select and "\$start <= $hour"
  db process {
    if {$msg1$greeting == $greeting} {
      set msg1 $greeting }
    if {$msg2$message == $message} {
      set msg2 $message }
    if {$rings$no_rings == $no_rings} {
      set rings $no_rings }
   }
  set greeting "$msg1 $msg2"
```

† SPARCstation is a trademark of Sun Microsystems

The plan is to choose two different greeting messages, to be played consecutively, and the number of rings to wait until the telephone is answered. The first two set commands figure out the current day of the week and hour of the day. The variables `msg1` and *msg2,* which will contain the two greeting messages, start off empty, as will `rings.` The `db select` command evaluates its argument as a TCL expression for each line of the configuration file, with the TCL variables corresponding to each field name containing the value for the current row. Only those rows for which the expression is true remain selected. After the three `db select` commands are finished, only those rows in the database that match the current time and day will be selected.

The code in the `db process` command gets executed once for each selected row in the database. The first selected row in which either of the messages or the number of rings is specified, causes the appropriate value to be filled in. The final `set` command sets the greeting message to the concatenation of the two message files. The message files contain pre-recorded messages.

The sample applications shown so far have been simple, and chances are good that they could be typed in and work on the first try. More complex applications can be debugged interactively using the built-in debugging features of **P**hone**S**cript. **P**hone**S**cript is normally run in batch mode, by running an existing **P**hone**S**cript program. **P**hone**S**cript may also be run interactively, like the *shell*. The user is prompted for commands from the keyboard. This is a useful way to test fragments of an application. This can be a tedious way to develop entire applications, however. The **P**hone**S**cript command `debug` causes **P**hone**S**cript to enter interactive mode from within a batch file, accepting TCL and **P**hone**S**cript commands from the keyboard. If the TCL *variable* `debug` is set, then **P**hone**S**cript will automatically enter interactive mode when a **P**hone**S**cript command fails. The error can be corrected interactively by retyping the command, and batch mode resumed by typing `exit` from the keyboard. The use of `debug` can be further enhanced with a TCL procedure such as TCL procedure `edit_proc,` shown in Figure 9, that can invoked interactively with the name of a (presumably errant) procedure. The `edit_proc` procedure writes the procedure provided as an argument into a file, starts up a text editor with that file, then reads the procedure back into the running **P**hone**S**cript program.

Figure 9: Interactively Edit a **P**hone**S**cript Procedure

```
# Interactively edit a procedure

proc editproc {name} {
  global pid argv

  if {[info procs $name] == ""} {
     echo "$name not found"; return }
  set file /tmp/$name.$pid.tcl
  set fd [open $file "w"]
  set args [info args $name]
  set body [info body $name]
  puts $fd "# $argv(0) [prdate {%D %T}]0
  puts $fd "proc $name $args $body"
  close $fd
  exec vi $file < /dev/tty > /dev/tty
  uplevel "source $file"
  }
```

† SPARCstation is a trademark of Sun Microsystems

Using this facility, the core of an application can be written in advance, and the remainder while the application is running. A missing feature will cause an error, interactive mode will begin, the new feature can be added, and execution of the program resumed.

As a final debugging aid, each **P**hone**S**cript command is assigned a letter that causes it to display various diagnostic and debugging information, when that letter is contained in the value of the *debug* variable. The various types of diagnostics may be enabled or disabled simply by changing the value of `debug.`

4. Related Work

The BerTel computer controlled telephone switch [10] [11] demonstrated that telephone and computers can talk to each other. The system also pointed out there needs to be a better way of constructing new telephone based services. The Expect [12] language shows how interactive programs can be tied together with a procedural language that has a built in notion of timeouts as expected conditions. The TCL embeddable command interpreter proved to be easy enough to use, that its simpler to build the right tool for a particular task, than it is to force the wrong one into service. Finally, the availability of multi-media mail transport facilities [13] and multi-media **e**mail user interfaces [14] provide **P**hone**S**tation with a connection into the workstation environment.

5. Summary and Conclusions

In addition to assorted answering machine programs, **P**hone**S**tation has been used to construct a directory assistance service, a survey system, an automatic scheduling program, and a fax document server. The survey system, used to evaluate the quality of the ORATOR® speech synthesizer under varying speaking parameters[15], was constructed in **P**hone**S**cript by a summer student with no prior UNIX experience in a couple of weeks. **P**hone**S**tation is in continuous service as part of the multimedia **e**mail system, providing users without audio capabilities on their workstations the ability to generate audio **e**mail, and to receive the audio portions of multi-media **e**mail messages over the telephone.

The ease of incorporation of TCL into the **P**hone**S**tation environment for the creation of **P**hone**S**cript is a tribute to the design of TCL. New flow control constructs, such as the **P**hone**S**cript event handling, and the extension of the `continue` semantics within the `play` command were easy to implement, eliminating the need to build a special purpose command interpreter for **P**hone**S**tation. As new technologies become available, such as speech recognition, new **P**hone**S**cript commands can be added to extend its functionality while maintaining the existing framework. Several applications, including the automated directory assistance system, were written twice, once in C using the library

interface,
and again directly in **P**hone**S**cript. In all cases the **P**hone**S**cript applications were shorter, easier to write, and took less time to get working than the C language versions. The interactive response of both versions is essentially the same.

Phone**S**tation demonstrates that the telephone, which has been traditionally ignored as a component of a workstation environment, can be integrated successfully, and provides not only better control of the telephone than an ordinary telephone, but extends the capabilities of the workstation as well.

6. References

1 Goldberg, A., (ed) *A History of Personal Workstations* ACM Press, 1988 pp 316.

2 ZiLOG. *Z8 Family Design Handbook* Campbell Ca., 1989

3 Ousterhout, J. *TCL: An Embedded Command Language* USENIX Winter conference proceedings, January, 1990, pp 133-146.

4 Spiegel, M.F., Macchi, M.J., and Gollhardt, K.D. *Synthesis of names by a demisyllable-based speech synthesizer (ORATOR®)*, Eurospeech ’89 Conference Proceedings, September 26-28, 1989, pp 117-120.

5 Kaiser, J. *Algorithms for Second Order Recursive Digital Filter Design* Unpublished Memorandum, 1992.

6 Sun Microsystems *Soundtool Manual Page* SunOS Reference Manual March 1990, pp 1782-1784.

7 Borenstein, N., and Freed, N. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies* RFC 1341, Internet Advisory Board, June, 1992.

8 Sun Microsystems *Sendmail Manual Page* SunOS Reference Manual March 1990, pp 2100-2102.

9 Bellcore *CLASS Feature: Calling Number Delivery* Bellcore Technical Reference TR-TSY-000031 Issue 3, January, 1990

10 Redman, B. *Who Answers Your Telephone When You’re in the Information Age?* Summer 1985 USENIX Conference Proceedings Portland, OR, pp 569-576.

11 Redman, B. *A User Programmable Telephone Switch* Unpublished internal Bellcore memorandum, April, 1988.

12 Libes, D. *Expect, Curing those Uncontrollable Fits of Interaction* USENIX Summer conference proceedings, June 1990, pp 11-15.

13 Borenstein, N. *Multi-media mail from the bottom up or Teaching Dumb Mailers to Sing* Winter Usenix Conference proceedings, January, 1992, pp 79-91.

14 Uhler, S. *MUI, a Window Based User Interface for Multi-Media Mail* Proceedings of the Bellcore/BCC Symposium on User Centered Design, Bellcore Special Report SR-OPY-002130, November, 1991, pp 171-175.

15 Macchi, M. J. et al *Intelligibility and Naturalness as a Function of Speaking Rate and Word Boundary Strength in the ORATOR® System* Presented as a talk at the IEEE Workshop on Interactive Voice Technology for Telecommunications Applications. October 19, 1992

7. Author Information

Stephen Uhler joined Bell Communications Research at its inception in 1984, where he is a Member of the Technical Staff in the Computer Systems Research division. He has worked on computing environments and user interfaces for much of that time, and is the author of the MGR window system. Before joining Bellcore, Stephen was a Member of the Technical Staff at Bell Laboratories in Whippany N.J. where he worked on user interface management systems. He received an M.S. degree from Case Western Reserve University. Stephen can be reached via electronic mail at: sau@bellcore.com -or- uunet!bellcore!sau