Amanda Portal System Reference Manual

Version 1.27E

The Amanda Company, Inc.

January 9, 2003

Contents

Preface

In all of the function definitions in this document, if there is a '*' next to the function name, you can execute the function when you aren't logged in. Otherwise, you must be logged into the system. In a related vein, many functions have the notion of a current box. When you don't give the `-box` option, they operate on the current box. When you are not logged in, there is no notion of a current box and these functions require the `-box` option.

Commands syntax is *verb noun*. Normally, if a command can return one or more results, the '`s`' suffix is left off the command (*e.g.*, `set_box_setting` rather than `set_box_settings`). This makes it easier for the programmer to remember the name of the command. Commands that require a box parameter (*e.g.*, `is_box`) do not use the `-box` syntax; the box is simply given as a required parameter. The `-box` syntax is for options only.

A fundamental theme in Amanda Portal is the notion of the box hierarchy and permissions related to the box hierarchy. Boxes create other boxes in an ancestral tree structure. Generally, unless specified otherwise, boxes only have permission to modify their box settings and the settings of their descendents. They cannot modify the setting of their proper ancestors.

Every command in this document can get a "usage" error if you call the command wrong. In this case, `errorCode` will be set to `USAGE` and `interp->result` will be the usage of the function.

Many commands in this system may process half an argument list before discovering an error. This means that half of the arguments changed values in the database and the other half didn't. Care is taken to alleviate this problem as much as possible though, for example, by checking the syntax of the arguments *before* executing transactions against the database.

Chapter 1

Introduction

The Amanda Portal system consists of a core set of functionality, which defines basic building blocks from which a variety of call processing systems can be built. A variety of plug-in modules, in the form of Windows DLLs, can be installed which augment the functionality of the core. By using plug-in DLLs, a system can be configured to load only the code and features that it requires. Since third parties can also develop these DLLs, there is a great deal of flexibility and extensibility in the system. The core and the plug-in modules together define an API which is similar to that of an operating system. They define a security model, provide networking features, etc.

Above this "API" layer is a scripting language, the Tool Command Language (Tcl), which is now familiar to over one million programmers world-wide. The scripting language serves two different purposes:

1. It acts as a "glue" to tie together the functionality of the different loadable modules (DLLs). As such, it acts as a very high level language for voice processing needs.
2. It allows quick and easy customization of an existing system, or rapid development of whole new voice processing solutions. The entire Amanda@Work.Group Telephone User Interface (TUI) consists of only a few thousand lines of Tcl code.

This architecture has several advantages:

- It separates functionality (*how* a particular task is implemented) from behavior (*why* a particular task should happen, such as a certain sequence of DTMF digits were input by a user).
- Because Tcl is the *only* interface to the system, network and telephone clients use exactly the same interface to accomplish tasks within the system. There is exactly one function which can be used to send voice mail to a mailbox, for instance; this function is used by all types of clients.
- Because the functionality is implemented in C++, it operates very efficiently, while the telephone user interface is scripted because this is where a great deal of flexibility is required. The TUI Tcl script is basically just implementing flow of control across all the possible functional calls (record, play, get digits, send voice mail, etc.).
- Since all the functionality is implemented in C++ and is not maliable, it is possible to implement a secure system. Users are allowed to execute any Tcl commands that are made available to them but cannot side-step the security model that is built into the system at the lower level.

- This architecture allows the TUI to be completely flexible without having to reimplement any functionality at lower levels. For instance, it would be quite possible to build a speech-recognition based voice mail system which would be quite different from the DTMF-based, Amanda@Work.Group-like interface that is currently provided. The menuing system would presumably be much "flatter," but the underlying features such as sending and listening to voice mail, changing mailbox settings, etc., would remain exactly the same at the C++ level.

1.1 Core Features

The core module implements a number of fundamental objects in the system: mailboxes, MultiMedia Objects (MMOs), a number of databases (the MMO lookup table, the voice mailbox database, etc.), an autoscheduler, a notifier and outbound call job management system, etc.

As part of the system's security model, mailboxes are arranged in a hierarchy. Every mailbox has a "parent" mailbox which is the one which created it. A mailbox may be given the right to create subordinate, or children, mailboxes. A top-level mailbox's parent is itself, since no mailbox created it, and such a box is referred to as a *superbox* in this manual. Such a mailbox is given certain special privileges in Amanda Portal, similar to the "root" user on a Unix system. Unlike Unix, however, there can potentially be more than one top-level mailbox (a "forest" of mailbox trees), although it is unlikely that this feature will be needed in practice.

Mailboxes have several components: they have some settings (the list of settings is extensible to make development of new applications more flexible), they have a set of privileges which have known semantics to the core, they may have messages which they have received, and the messages in turn may have zero or more MMOs associated with them. Boxes may "publish" MMOs for others to use, such as greeting recordings. Boxes can establish autoschedule and notify records which cause the system to execute Tcl code on their behalf when certain events happen or certain times arrive.

1.2 Tcl As An Extensible Glue Language

Tcl makes a good embedded systems language because it is high-level, easy to extend the language by adding C functions as Tcl primitives and has security mechanisms built in using the nested interpreter model and "code injection"[1]. Nested interpreters give you an operating system like "system call" interface where you write your "system calls" in Tcl. In this model, the system calls always exist in your interpreter. If you are not allowed to perform a call, the system call returns an error. The master interpreter operates in "kernel" mode and the slave interpreter operates in "user" mode.

Code injection works slightly differently. The protected calls you need to use don't check whether you have permission to execute them; instead, the calls don't exist in your interpreter until a specified event happens (such as you log in). At this point, the call is inserted into your interpreter and you can use the new calls. When you log out, the calls are removed from your interpreter.

Tcl is a Lisp like language so writing code with it is quite "functional." You often write code like

```
eval modify_elements [get_elements handle]
```

[1] A tAA defined term.

where `get_elements` returns the operands to operate on as a Tcl list and `modify_elements` operates on those operands. Tcl also helps extensibility by being interpreted. This means that you can add code on the fly to your program and modify the behavior of the code shipped with the system. Tcl also has associative arrays built in; these are the regular Tcl array variables. Internally, they are implemented as hash tables so lookup on them is very speedy. Multi-dimensional arrays can be simulated in Tcl by using concatenated indices in regular arrays.

1.2.1 An Introduction to Tcl

Safe interpreters are simply regular interpreters with certain "unsafe" calls removed; that is, they are interpreters where some unsafe code has already been "dejected." All Tcl interpreters in Amanda Portal are of the "safe" variety, and as such, they have only the following standard Tcl commands: `append`, `array`, `break`, `case`, `catch`, `concat`, `continue`, `error`, `eval`, `expr`, `for`, `foreach`, `format`, `gets`, `global`, `if`, `incr`, `info`, `interp`, `join`, `lappend`, `lindex`, `linsert`, `list`, `llength`, `lower`, `lrange`, `lreplace`, `lsearch`, `lsort`, `proc`, `regexp`, `regsub`, `return`, `scan`, `set`, `split`, `source`, `string`, `switch`, `time`, `trace`, `unset`, `uplevel`, `upvar`, and `while`.

This subsection gives a brief description of each of these commands. Those who are already familiar with Tcl should skip ahead to the next section. There are many books available which cover the Tcl language, including *Tcl and the Tk Toolkit,* by Tcl's author, John Ousterhout, and *Tcl For Dummies,* in the famous "Dummies" series. While those books should be consulted for in-depth examples, they also cover many more features of Tcl, plus the Tk graphical extensions, which are not used in Amanda Portal.

Tcl Variables

Programming languages contain two fundamental elements: variables and procedures. Object-oriented languages combine these two features into *objects* (naturally). Variables in Tcl have three attributes:

name The variable's *name* allows access to its contents.

scope The *scope* of a variable determines how long it will live. When it goes *out of scope,* a variable is automatically destroyed.

value Finally, each variable has a value.

In many programming languages, variables have a fourth attribute, a *type,* which determines the kind of data that the variable can have as its value, the size of the date (range of values, length of a string, etc.). In these languages, variables are *declared,* and a variable can exist without having a defined value.

Tcl's variables, like those of Amanda@Work.Group, do not have a specific type. Internally, they are all represented as strings and are converted on-the-fly as needed to other types (floating point, integer) when necessary. This feature makes Tcl programming easier. A variable springs into existence the first time it is used. This also means that it's impossible to have a variable which does not have any value.

To create a variable in Tcl, you use the `set` command, like this:

`set` *variable value*

For example, to set the variable `myname` to `Tim`, you would simply type

```
set myname Tim
```

To access the value of a variable, put a dollar sign in front of its name. For sample, to copy the value of the variable `myname` to a second variable called `firstname`, you would type

```
set firstname $myname
```

This process is called *variable substitution:* the name of the variable is replaced by its value.

Each executable statement in Tcl consists of a single command line, which is terminated with a newline or a semicolon. For instance,

```
set x 14
set y 27
```

could just as easily be written

```
set x 14; set y 27
```

Tcl evaluates each command line using a two-step process: *parsing* and *execution.* During parsing, the command line is divided into space-separated *arguments,* then variable substitution and other processing is performed on the arguments. At this point, Tcl doesn't attach any meaning to the arguments themselves; it's just doing simple string substitutions.

In the execution step, the first argument of the line is treated as a command name to be run. The other arguments on the command line are passed on to the command, which will apply meaning, if any, to them. The command will return a string value as the result of its processing. In each of the examples above, the command name was `set`.

What would happen if you tried to execute the command

```
set name Tim Morgan
```

This is a problem because `set` will see three arguments rather than the two it's expecting (a variable to set and a value to set it to). Just as in Amanda/DOS, we can overcome this problem by using quotation marks to "hide" the space from the parser:

```
set name "Tim Morgan"
```

Now suppose that we have two variables, `firstname` and `lastname`, and what we need is the person's full name. We can achieve this by writing

```
set name "$firstname $lastname"
```

On the other hand, if we write

```
set name {$firstname $lastname}
```

then no variable substitution will be performed. We can also mix and match:

```
set salary "$firstname $lastname \$30,000"
```

This would expand the values of *firstname* and *lastname*, but it would leave the dollar sign before 30,000.

Be careful of the following types of commands:

```
set x 4
set y x
set z x+y
set $y 5
```

The first sets `x` to `4`, as expected, but the second command sets `y` to the string `x`, not to the *value* of `x`.

Remember that Tcl does not associate any meaning with the arguments as it parses them. Therefore, the third command simply sets variable `z` to the string `x+y`. We'll learn how to accomplish addition shortly.

The last command sets `x` to 5, not `y`.

We have already seen an example of *backslash substitution* when we used `$` to produce a literal dollar sign character in a string. Many of the backslash substitutions are the same as those in Amanda/DOS:

Symbol	Meaning
`\a`	Audible alert (bell)
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline
`\r`	Return
`\t`	Tab
`\v`	Vertical tab
`\`*ddd*	Octal value
`\x`*hh*	Hex value
`\`*newline*	Single space
`\$`	Dollar sign
`\\`	Backslash
`\[`	Open square bracket

Tcl Lists

A Tcl *list* is an ordered collection of strings. In its simplest form, a list is simply a string, with spaces separating the list's elements. In this case, the elements cannot contain spaces.

Lists are usually written surrounded by curly braces. Sublists, or elements with spaces, are denoted by inner braces. For example,

```
set x {a b {c d e} f}
```

sets `x` to a list containing four elements. The third element is the list (or string) `c d e`.

Tcl contains a number of built-in functions for manipulating lists.

Built-In Commands and Procs

So once the command to execute has been determined, and Tcl knows what the arguments are, what happens next? Tcl will look up the command name to verify that it's valid. If it isn't, then an error is generated.

If it is a valid command, then there are two possibilities: the command is a "built in" one or it's a "Tcl procedure."

Commands (procedures, or "procs") which you define in Tcl are indistinguishable from the built-in commands, except that the built-in ones are faster and can access operating system and device driver level calls.

A "built in" Tcl command has its underlying implementation in C. In this case, Tcl invokes the corresponding C procedure and processing of the command is done in compiled code. The `set` command is a built-in, for example.

Tcl defines a number of built-in commands, and applications using Tcl can define additional built-in commands. Not surprisingly, Amanda Portaldefines quite a few application-specific commands which will be covered in the Amanda PortalProgramming class.

So far, we've seen two kinds of substitutions which happen automatically to commands: variable and backslash. We also know that all Tcl commands (procedures) return a value. This leads to the third type of substitution, *command substition.* To illustrate it, we'll introduce another built-in command, `expr`.

Expressions

The `expr` command concatenates all its arguments into one long string, then it evaluates that string as an expression, and it returns the resulting value as its procedure value (remember that all procedures return a value in Tcl).

Here are some example interactions with Tcl:

```
% expr 2*3
6
% set x 4
4
% expr $x * 7
28
% expr ($x * $x) / 2
8
%
```

Now we're ready to discuss the third type of substitution, *command substitution.* Recall that every command returns a string result. Command substitution allows us to replace a complete Tcl command with its result for use in an enclosing command. To use it, just enclose a command inside square brackets: `[]`. For example:

```
% set x 4
4
% set y 7
7
% set z [expr $x * $y]
28
%
```

Arrays

So far, the variables we've used have been *simple variables,* also called *scalar.* Tcl also provides another kind of variable, called an *associative array.* An associative array is a collection of different values which are indexed by a string (the element name). This is a very powerful mechanism, and associative arrays are used extensively in Amanda Portal.

Array elements thus have two names: the name of the array plus the name of the element. The element name is enclosed in parentheses. It may be a constant or a variable (or a combination), so long as it doesn't contain a space. If the element name has a space, then the entire variable name must be quoted to "hide" the space.

For example, we could collect information on a person this way:

```
set person(name) "Tim Morgan"
set person(age) 37
set person(address) "23822 Brasilia Street"
set person(city) "Mission Viejo"
set person(state) California
set person(zip) 92691
```

On the other hand, if we have information about a number of people, we might use one array per attribute, and use the names as the indices:

```
set age(Tim) 37
set address(Tim) "23822 Brasilia Street"
set city(Tim) "Mission Viejo"
set state(Tim) California
set zip(Tim) 92691
set age(Siamak) 36
set address(Siamak) "26321 Eva Street"
set city(Siamak) "Laguna Hills"
set state(Siamak) California
set zip(Siamak) 92656-3108
```

We can use the values in the array pretty much just as with simple variables:

```
% set foo $age(Tim)
37
% set name Tim
Tim
% set foo $age($name)
37
% set foo [expr $age(Tim) + $age(Siamak)]
73
%
```

Besides `set`, Tcl defines a few other built-in functions for working with variables:

`append` Appends one or more values to a variable.

`incr` Adds a value to an existing numeric variable.

`unset` Deletes one or more variables.

`array` Returns various information about array variables.

The syntax for the `append` command is as follows:

`append` *varname value ...*

If *varname* is an existing variable, then *value* will be appended to it, as will any additional values.

If *varname* didn't exist previously, then it is set to *value* as if the `set` command had been used.

Like `set`, `append` returns the new value of *varname* as its result.

The `incr` command is exactly like the `+` token in Amanda/DOS. It allows you to do an arithmetic addition to an existing numeric variable. The value added may be positive or negative, and it defaults to 1.

```
% set i 1
1
% incr i
2
% incr i 2
4
% incr i -3
1
%
```

The variable must exist and have an integer value. The `incr` function returns the new value as its result.

The `unset` command is used to delete a variable and its associated value(s). If an array variable is deleted, then all of its elements are deleted. The general syntax for the command is:

`unset` *var ...*

The `unset` command always returns an empty string as its result:

```
% set age(Tim) 37
37
% set age(Siamak) 36
36
% unset age
%
```

The Array Command

The `array` command allows you to obtain information about a Tcl array. The general syntax is:

`array` *cmd name args*

where *cmd* is a special keyword for the `array` command. Some of the available keywords are:

`names` *ary* Returns a Tcl list of the names (valid indices) for array *ary.*

`size` *ary* Returns the number of elements in array *ary.*

`exists` *var* Returns whether *var* exists and is an array.

`get` *name pattern* Returns a list of pairs of element names and values from the array *name.* If *pattern* is specified, then only those elements whose names match *pattern* will be returned. A list of name and value pairs is called an *a-list.*

`set` *name list* Sets elements of the array *name* from *list* which should be of the same form as is returned by `array get`.

Normally, the `array` command will be used in conjunction with *command substitution* along with some other commands which we'll examine next week. In the mean time, here are some standalone examples of `array`, using the `age` array we defined earlier:

```
% array size age
2
% array names age
Tim Siamak
```

Notice that the result of the `array names` command will list the names in arbitrary order. If you want it sorted a particular way, you can use the `lsort` command discussed on page 16.

Tcl Expressions

Expressions combine values with operators to produce a new value. Simple examples follow everyday arithmetic rules. For example:

```
% expr (8+4) * 6.2
74.4
%
```

In addition to the usual binary operators `+`, `-`, `*`, and `/`, there are a number of other important unary and binary operators (and there are some less important ones which we won't mention). There's also `%`, which computes the remainder from a division (this is the "mod" or "modulo" operator).

Most operators are called *binary* because they work on two values. For instance, `+` is a binary operator since it is used to add two different values, which are written on either side of it.

Unary operators, in contrast, are operators which work on a single value which comes after the operator. The most familiar one of these is *unary minus,* which negates the value of its operand.

```
% set x 4
4
% set y [expr -$x]
-4
%
```

The other important unary operator is *logical not,* which is written `!`. It changes true values to false, and false values to true.

In Tcl, as in C, logical values are represented as numeric values. A false value is represented as 0, while a true is any non-zero value. Therefore,

```
% set x 4
4
% set y [expr !$x]
0
% set x [expr !$y]
1
%
```

Relational operators are binary operators to compare one value to another. They all work on integer, real (floating point), and string values.

Syntax	Result
`a<b`	1 if $a < b$ else 0
`a>b`	1 if $a > b$ else 0
`a<=b`	1 if $a \leq b$ else 0
`a>=b`	1 if $a \geq b$ else 0
`a==b`	1 if $a = b$ else 0
`a!=b`	1 if $a \neq b$ else 0

Often you want to combine the results of more than one logical test. The *logical operators* allow you to do so. They follow the same syntax as in the C language.

Logical *and* is represented as `&&`. The expression `a&&b` is true (1) if both the expressions a and b are non-zero, and zero otherwise.

Logical *or* is represented as `||`. The expression `a||b` is true (1) if *either* of the expressions a or b is true, and zero otherwise.

As in C, these operators are *short-circuit*: the second operand is not evaluated if the first shows the result of the whole thing.

Here are some examples of logical operators:

```
% set a 1
1
% set b 0
1
% expr a&&b
0
% expr a || b
1
% expr b || a
1
% expr b && exp(sin(sqrt(2.0))) > 4
0
% expr a || exp(sin(sqrt(2.0))) > 4
1
%
```

In the last two examples, function `foo` is *not* called, because the result can be determined from the value of the first argument alone.

List Commands

Tcl defines quite a few commands for operating on lists. We're going to look at the following ones:

`lindex` Retrieves individual elements from a list.

`llength` Returns the length of a list.

`lsort` Returns a new list resulting from sorting an existing list.

`lappend` Appends new items to an existing list (variable).

`split` Returns a list by spliting up a string at each point where a given substring occurs.

The `lindex` function returns as its value an element of a list. The general syntax is:

`lindex` *list index*

The *index* must evaluate to an integer. The first element of the list is number `0`. For example:

```
% set mylist {a b {c d e} f}
a b {c d e} f
% lindex $mylist 1
b
% lindex $mylist 2
c d e
% set x {a b c d e}
a b c d e
% lindex $x 4
e
%
```

The `llength` function simply returns the number of elements in a list. Its syntax is

`llength` *list*

Continuing from the previous example:

```
% llength $x
5
% llength $mylist
4
% set len [llength [lindex $mylist 2]]
3
%
```

The `lsort` command returns a new list which is created by sorting an existing list. The syntax is:

`lsort` *flags list*

The *flags* parameter can be a combination of a type and a direction. The types are:

`-integer` Treat the list elements as integers.

`-ascii` Treat the list elements as strings (default).

`-real` Treat the list elements as floating point (real) numbers.

The direction can be one of the following:

`-increasing` Sort the items in ascending order (default).

`-decreasing` Sort in decending order.

Here are some examples of `lsort`:

```
% set x {22 12 9 97}
22 12 9 97
% lsort $x
12 22 9 97
% lsort -integer $x
9 12 22 97
% set sorted_list [lsort -integer $x]
9 12 22 97
%
```

The `lappend` command appends one or more values to a variable as list elements. It creates the variable if it doesn't already exist. It differs from `append` because with `lappend`, the items are appended with separating spaces and will be given braces where necessary to keep items separate; `append` simply appends each string as-is.

The general syntax is:

`lappend` *varname value ...*

The `lappend` function returns the new value of *varname* as its result value.

Here are some examples:

```
% set x {a b c}
a b c
% lappend x d e f
a b c d e f
% lappend x "Tim Morgan" "Carl Doss"
a b c d e f {Tim Morgan} {Carl Doss}
% llength $x
8
%
```

Notice that in the second `lappend` command, two values are being appended. This is reflected in the result of `llength`: there are eventually 8 items in the list.

The `split` command creates a new list by splitting up a string. You can specify a set of separator characters as the second argument, or let it default to standard white-space characters.

```
% split "abc/def/ghi" "/"
abc def ghi
% split "abc.def/ghi" "/."
abc def ghi
%
```

---

* `linsert`

Description

`linsert` can be used to insert items into an existing list variable at a particular index.

---

* `lrange`

Description

`lrange` can be used to extract a range of items from a list (returning a new list).

---

* `lreplace`

Description

`lreplace` can be used to create a new list from and old list with a range of items replaced by a new set of items (not necessarily the same number of items).

Eval

One final note on parsing. Sometimes you have a list of items in a variable, and you want to pass that list as arguments to some command, such as lappend. Here's an example of the problem:

```
% set mylist {a b c}
a b c
% set additional_items {d e f}
d e f
% lappend mylist $additional_items
a b c {d e f}
%
```

What we really wanted was to get a resulting list of `{a b c d e f}`, not `{a b c {d e f}}`. The `eval` command lets us solve this and other similar problems.

The `eval` command takes one or more arguments and concatenates them into one long string. It then evaluates (executes) this string as as a Tcl command. Let's look at how we'd use `eval` in the previous example:

```
% set mylist {a b c}
a b c
% set additional_items {d e f}
d e f
% eval lappend mylist $additional_items
a b c d e f
%
```

Why does this work? The `eval` command sees three arguments: `lappend`, `mylist`, and a string "`d e f`" (the quotation marks aren't part of the string). It appends all of these strings together, with separating spaces, arriving at the new command string "`lappend mylist d e f`" (again, the quotation marks aren't part of the string). Then `eval` executes the command, and `mylist` gets three new elements instead of one sublist element.

Control Flow

Normally, a program is executed sequentially, one statement after the previous one. Most programming languages, including Tcl, provide two basic ways to vary this.

Branching means skipping to or past some statements. This is provided with the `if` and `switch` commands.

Looping allows a set of statements to be repeated zero or more times. This is provided by the `while`, `for`, and `foreach` commands, along with their helper commands `break` and `continue`.

The general syntax for the `if` statement is:

`if` *test1 body1 [*`elseif` *test2 body2 . . .] [*`else` *bodyN]*

The *testn* values may be any expression which evaluates to a Tcl boolean value (zero or non-zero). Typically, they will include relational operators. The *bodyn* values may be any Tcl statement or list of statements. Remember that statements may be separated by semicolons or RETURNs, and uninterpreted lists may be written using curly braces, and they "hide" any embedded RETURNs.

This means you can write statements like:

```
if {$x < 0} "set x 0"
if {$x < 0} {set x 0; set y 7}
if {$x < 0} {
    set x 0
    set y 7
}
```

But you may *not* write

```
if {$x < 0}
{
    set x 0
}
```

The most common way to write `if` statements is like this:

```
if {$x < 0} {
    ...
} elseif {$x == 0} {
    ...
} elseif {$x == 1 && $y == 7} {
    ...
} else {
    ...
}
```

The `switch` statement allows you to match a string against a number of patterns, and if a match is found, to execute some code. The pattern matching may be one of:

`-exact` The strings must match exactly.

`-glob` Wildcards `*` and `?` can be used, just like the DOS `DIR` command; this is the default).

`-regexp` Regular expression pattern matching is applied—see the manual for details.

Suppose that you want to test `$x` to see if it's any of the strings `a`, `b`, or `c`. You could do this with `if`:

```
if {$x == "a"} {
    incr t1
} elseif {$x == "b"} {
    incr t2
} elseif {$x == "c"} {
    incr t3
}
```

With the `switch` statement, you can write this more clearly:

```
switch $x {
    a {incr t1}
    b {incr t2}
    c {incr t3}
}
```

While `while` statement's syntax is:

`while` *test body*

Here's an example of reversing a list `a`, putting the result into list `b`:

```
% set a {1 2 3 4 5 6}
1 2 3 4 5 6
% set b {}
% set i [expr [llength $a] - 1]
6
% while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
-1
% set b
6 5 4 3 2 1
%
```

The `for` statement is usually used to perform a loop for a specified number of times, using a *control variable* to count the number of iterations. The Tcl syntax, like that of C, is more general:

`for` *init test reinit body*

The `for` statement executes the *init* statement. Then while the *test* is true, it executes *body* and *reinit.* Thus, `for` is really a `while` statement with a few other statements thrown into the mix. You can accomplish the same things with either statement. When a loop is controlled by a variable, then using `for` will result in more easily understood code.

To iterate variable `i` from 1 to 10, you would write

```
for {set i 1} {$i <= 10} {incr i} {
    set array($i) "This is item $i"
}
```

To reverse list `a`, putting the results in `b`:

```
set b {}
for {set i [expr [llength $a] - 1]} {
  $i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

The `foreach` statement is used very frequently in Tcl. It iterates a control variable through each item in a Tcl list, executing a body of statement(s) each time. So to reverse a list, one could write

```
set b {}
foreach i $a {
    set b [linsert $b 0 $i]
}
```

To reset each element of an array to an empty string, one could write:

```
foreach i [array names a] {
    set a($i) ""
}
```

Sometimes you want to leave a loop or repeat a loop from somewhere in its interior. This is the purpose of the `break` statement. Suppose you know that `ary(n)` is equal to 7, for some value of n. To determine n, you could write:

```
foreach i [array names ary] {
    if {$ary($i) == 7} {break}
}
# At this point, $i is the index into
# "ary" for the value "7"
```

Using `continue` within a `for` causes the *reinit* code to be executed. With both `for` and `while`, the *condition* must still be true before the *body* will repeat again.

Tcl Procedures

A *Tcl procedure* is a command defined solely in Tcl. It may use other Tcl procedures or built-in commands to do its work. You may (re)define Tcl procedures whenever you wish using the `procproc` command. Tcl procedures have three attributes:

name This is, of course, the command name which will invoke the procedure.

arg_list This is a list of the *formal parameters* of the procedure. That is, it's a list of the names by which this procedure will refer to its arguments. The number of arguments passed on the command line must normally match the number of arguments in the list.

body The body is a list of Tcl commands to be executed. It's just like the bodies of the `if`, `while`, etc., statements.

---

* `proc` *name arg_list body*

Description

The `proc` command is used to (re)define a procedure. You always give it three argments: the name, the argument list, and the procedure body. If a procedure (or a built-in) with the specified name already exists, then the old one is deleted and replaced by the new one.

Procedures can return an explicit value by using the `return` statement. If the last statement of the procedure body is not `return`, then the result of the procedure will be the result of that statement.

That means that the following two procedure definitions produce equivalent results. The first one is slightly slower, but it's clearer to someone reading the code that it's intended to return the sum of the two arguments.

```
proc plus {a b} {return [expr $a + $b]}
proc plus {a b} {expr $a + $b}
```

Either way, you can use the new `plus` command as follows:

```
% proc plus {a b} {expr $a + $b}
% plus 2 3
5
%
```

Notice that the arguments which are written on the command line (2 and 3) are *passed* to the procedure and it "knows" them by the names `a` and `b`.

```
% plus 2
no value given for parameter "b" to "plus"
% plus 2 3 4
called "plus" with too many arguments
```

We can also write a procedure which takes a variable number of arguments and returns the sum of all of them, using the keyword `args`, whose value will be the list of the arguments passed:

```
% proc sum {args} {
    set result 0
    foreach i $args {incr result $i}
    return $result
}
% sum 1 2 3
6
%
```

Notice in the previous example that we used a "helper" variable called `result`, inside procedure `sum`. Whenever the `set` or related commands is used to create a variable inside a procedure, that variable is *local* to that procedure. That is, its value is independent of any other variables named `result` in other procedures, and when the procedure returns, this particular variable `result` and its value will be destroyed. This process is called "going out of scope"—once no code can possibly access that variable, then there's no need for it any more.

Thus, by default variables in Tcl are *local.* It is good coding practice to avoid the use of global variables whenever possible.

Global Variables

As we've already seen, variables can exist outside the scope of any procedure. These variables are called *global,* and they can be accessed from within procedures by declaring them with the special keyword `global` prior to accessing them.

Here's an example of a procedure accessing a global variable `x`. It accepts one argument `a`, adds it to the global variables `x` and `y`, and returns that result.

```
% set x 4
4
% set y 5
5
% proc foo {a} {
    global x y
    return [expr $x + $a + $y]
}
% foo 3
12
% set x 6
6
% foo 3
14
%
```

Local variables, as well as arguments, are created separately *each time* a procedure is invoked. This allows writing *recursive* procedures. Recursive procedures are procedures which may call themselves.

The classic example of a recursive procedure is one to calculate factorials. As you remember, by definition

$$n! = n \times (n-1) \times \ldots 1$$

so we can write a factorial function in Tcl as follows:

```
% proc fact {x} {
    if {$x <= 1} {return 1}
    expr $x * [fact [expr $x - 1]]
}
% fact 5
120
%
```

So far, every example of passing arguments to procedures has been *pass by value*: that is, the value of the variable is passed to the procedure, which knows that value by its own name. What if, instead, we want to write a procedure which can *modify* a variable which is passed as an argument?

To do this, or to pass an array as an argument, we need a way to pass arguments *by reference.* The `upvar` command allows us to do so. When declaring a procedure which is to receive an argument by reference, use a dummy name to mark the argument, then use `upvar` to assign this to a local name.

Notice that the variable `foo` is set to a new value by `myproc`:

```
% proc myproc {a} {
    upvar $a myvar

    set myvar 4
}
% set foo 3
3
% myproc foo
4
% set foo
4
%
```

Notice in this example that we can pass a whole associative array as an argument:

```
% proc array_size {a} {
    upvar $a myarray

    return [array size myarray]
}
% set age(Tim) 37
37
% set age(Siamak) 36
36
% array_size age
2
%
```

Errors and Exceptions

In older programming languages, all error conditions had to be handled by explicit checking throughout the code. For instance, when you open a file, you have to check the return value to see if the open was successful, and if not, take some action.

An *exception* is something out of the ordinary which occurs as a program is running. For instance, if you divide by zero, an exception might be raised, and it would usually cause the operating system to abort the execution of your program. Even DOS detects divide by zero exceptions, for example.

Modern thinking on programming languages is that exception and error handling should be concentrated in a few places within a program rather than sprinkled throughout. This has several benefits: primarily, that the code which does the "normal" work is much easier to read and understand (and therefore more likely to work correctly), because it doesn't have all the exceptional code interspersed.

When a Tcl procedure or built-in wants to raise an exception, it can execute a variation of the `return` statement to return not only the normal return string (which would probably be an error message of some type), but also an exception (or return) code. A code of 0 is the normal, non-error return. Codes 1, 2, and 3 are used for exception, `break`, and `continue` events. Other codes can be defined by the application.

When a non-zero return code is returned, Tcl will keep returning the code up the call stack until it either reaches the top level or it reaches a procedure which is prepared to handle an exceptional return.

The Tcl command used to catch exceptional returns is `catch`. The easiest way to use it is

`catch` *script var*

This will execute the command(s) in *script.* If it and any procedures it calls all return normally, then processing simply passes to the subsequent command. Otherwise, the `catch` command returns as its value the error code, and *var* will be set to the error message (string) which was returned from the *script.*

Amanda Portal is designed to minimize the number of places that an application programmer will need to use `catch`.

Here's an example of using `catch`, plus the `error` which can be used to raise an exception:

```
proc foo {a} {
    if {$a != "bar"} {
        error "The argument was invalid"
    }
    return "Everything ok"
}

proc foo {} {
    set code [catch {foo baz} result]

    if {$code} {
        puts "Error received: $result"
        # Pass the error up the stack
        return -code $code $result
    } else {
        puts "Execution ok: $result"
    }
}
```

The `info` command is used to get various pieces of information from the Tcl interpreter. It has many different options. The ones which are likely to be of interest are:

args *proc* Returns the list of arguments for procedure *proc.*

body *proc* Returns the body of procedure *proc.*

commands *pattern* Lists the built-in and Tcl procedures whose names match *pattern.* If no pattern is specified, all names are returned.

exists *var* Returns true if *var* is currently a valid variable name. You might do this before trying to access its value under some circumstances.

procs *pattern* This is like `commands`, but it returns only procedures defined in Tcl (those matching *pattern* if it's specified).

tclversion Returns the version number of the interpreter.

vars *pattern* Like `commands`, but it returns a list of variables (those matching *pattern* if it's specified).

1.2.2 Tcl and Multithreading

Tcl originally did not support multithreading in any way. Because Amanda Portal has a need for multiple threads because so many things are happening at once, tAA added multithread safety to Tcl. Upcoming versions of Tcl will include Unicode support for multiple languages and multithread safety; when Tcl 8.1 is released, Amanda Portal will be switched to use that version.

The multithreading that was added only allows one thread per interpreter. This is a good thing because the global variables `errorCode` and `errorInfo` are per interpreter and should not be shared between threads. Interpreters (and the threads which are tightly bound to them) are spawned off whenever a new "task" needs to be executed. A "task" is spawned for some event such as a call coming in. When a call comes in, for instance, the core creates a new interpreter and a new thread to handle the call. The entry point in the Tcl code can vary depending on configuration parameters used by that DLL.

1.2.3 Tcl and State

Tcl allows you to store state in a number of places. You can store state globally, on the stack (local variables) and in arrays. (Sort of. Tcl treats the variable as a single value with respect to scoping but you can store aggregate information in the array.)

Tcl allows you to attach internal state to Tcl functions and interpreters and to install triggers on Tcl variables. Interestingly, internal Tcl state cannot be attached to Tcl variables. Tcl state is attached to functions through the `ClientData` parameter. This parameter gets set when you create the function. Internal Tcl state is attached to an interpreter with the `Tcl_SetAssocData()` and `Tcl_GetAssocData()` functions. Triggers (called "traces" in Tcl) can be attached to variables that fire off whenever the variable is read, written or deleted.

1.2.4 Assigning Scope to Procedures

Procedures in Tcl are not scoped. This is unfortunate because we want to make VP and LS device handles and MMO handles into functions and be able to call the functions in an object-oriented fashion like

```
    $vp play $mmo
```

which would play the sound associated with the MMO handle on the specified VP device. Since procedures are not scoped, the handles would not disappear if the stack returned. That is,

```
proc play_greeting {} {
    grab_res vp vp_proc     ;# Get a single VP device. Return new proc name.
    # Get greeting 1 from box 10. Return new proc.
    lookup_mmo -box 10 "Grt 1 English" mmo_proc
    $vp_proc play $mmo_proc ;# MMO specific data in ClientData of $mmo_proc.
}
```

would not delete the procedures named by the values of the variables `vp_proc` and `mmo_proc` when the procedure is done because procedures are not scoped.

We can get around this problem by noticing two things: variables are scoped and behavior can be attached to variables (through triggers) when variables are deleted. Therefore, to solve this problem, we instead create both a variable and a procedure when we create an LS, VP or MMO handle. The variable has a deletion trigger attached to it. When the variable leaves scope, the deletion trigger automatically deletes the procedure, giving the procedure scope semantics.[2] The variable created is special: the Tcl variable has a deletion callback attached to it internally. Most Tcl variables don't. Also, the procedure handle created has internal state associated with it (through `ClientData`). This state stores such things as the internal C information associated with the VP device (for VPs) or the internal C information stored with the MMO (for MMOs). Most regular Tcl commands don't have internal state associated with them.

Let's explore this idea with an example. Suppose you want to play an MMO on a VP device. First you grab a VP device with the function `grab_res`:

```
grab_res vp vp_var   # Grab a single VP resource and put in vp_var
```

`Grab_res` does the following when it is called:

1. It looks for a VP device and allocates it to the caller. If no valid VP device is found or available, an error is returned.
2. It creates a new command, say "`_vp0`" and sets up that command to call an internal C function which implements the behavior of VP devices. It sets the `ClientData` parameter for this procedure to the internal state allocated for the new VP device.
3. It sets the variable `vp_var` to the name of this new command and it sets a deletion callback on this variable that will delete the command `_vp0` when the variable is deleted.

So if you now say

```
set vp_var        # Returns variable's value
```

you will get back the string `_vp0`. What you won't see is that the variable `vp_var` has a deletion callback assigned to it.

[2] This is also useful in C++. C++ has destructors that are fired off whenever a variable leaves scope, even when an exception is raised. These destructors can execute code you want executed every time a procedure on the stack returns. Suppose for example you are calling a procedure that needs to execute in a critical section. You also have two functions: `EnterCriticalSection` and `LeaveCriticalSection` that are put at the beginning and end of the procedure, respectively. The function looks like

```
int foo()
{
    EnterCriticalSection();
    ...do some processing...
    LeaveCriticalSection();
    return 0;
}
```

It is imperative that you call `LeaveCriticalSection` or your code will have a serious bug. To be absolutely sure that your code calls `LeaveCriticalSection`, you can have the system call it for you by putting the call to it in a deletion trigger of a dummy variable in the scope of foo. When foo finishes, the deletion trigger is called and `LeaveCriticalSection` is executed. For example,

```
int foo()
{
    CriticalSectionObj dummy; // Calls EnterCriticalSection on creation.
    // Deletion trigger for this variable calls
    // LeaveCriticalSection.
    ...do processing here...
}
```

1. `set r $v1` # Regular variable gets MMO handle.
2. `set v1 $r` # MMO handle set to regular variable.
3. `set v1 $v2` # MMO handle = MMO handle.

Figure 1.1: Different assignment scenarios.

Now suppose we want to play an MMO handle on this VP device. I'll assume we have an MMO handle stored in the variable `mmo_var` which references the command `_mmo0` (MMO handles work just VP handles). To play the MMO, we say

```
$vp play $mmo
```

The Tcl parser expands this to

```
_vp0 play _mmo0
```

and executes the command `_vp0` with the arguments `play _mmo0`. The `_vp0` command then does the following:

1. It sees that it is a `play` command and looks for MMO functions as arguments to the `play` command.
2. It looks up the command named `_mmo0` and extracts the `ClientData`. It now has the internal C information it needs to play this MMO. It also has the internal C information about the VP device because this was passed in the `ClientData` parameter when `_vp0` was called.

Variable handles created this way are immutable. That is, they can be created and destroyed but not assigned to. Let's examine why this is so. Suppose `r` is a regular variable and `v1` and `v2` are MMO handle variables. Figure 1.1 shows the three assignment scenarios that we need to be concerned with.

The first scenario cannot be prevented because Tcl doesn't provide triggers on arbitrary non-existent variables. That is, you can set a write trigger on the variable `r`, but how do you know *a priori* which new variables will be created? The semantics of operation 1 are to set `r` to the name of the procedure associated with the variable `v1`. No deletion trigger is copied to `r` so setting `r` in this manner will create a "handle" which doesn't obey the scope rules. The handle function may fail if `v1` leaves scope.

The second statement trashes an MMO variable. If this were allowed, the deletion trigger on `v1` would have to be executed on a write to delete the associated procedure and then `v1` would revert to a regular variable. This is not rocket science and could easily be accommodated.

The last assignment is the problem. In this scenario, an MMO handle is assigned to another MMO handle. What do we do here? Do we execute the trigger on `v1` to delete the procedure associated with it and then set up `v1`'s deletion trigger to be the same as `v2`'s? This won't work. Deleting `v1` will delete the procedure when it is still in use by `v2`. We would need reference counting in the procedure deletion logic. We could also make the operation turn `v1` into a regular variable by running its deletion callback and setting the variable to the name of `v2`'s procedure. This isn't very useful though and doesn't fit the clean definition of assignment. For this reason, handles are immutable. (By the way, read-only access on the variable is simply implemented by setting a "write" trigger which doesn't allow assignment.)

1.3 Control Flow

The Tcl code that defines the behavior of Amanda is called the Telephone User Interface or TUI. Tcl gives the TUI great flexibility and extensibility and is Amanda Portal's trump card. There are basically two ways we could have structured the control flow in Amanda when doing a "task":

1. Use a state machine. As each menu is presented, the user selects an option. When the option is selected, the next state is traversed and the next menu is presented. The advantage of this approach is that it uses a small stack and traversing from one unrelated state to another is easy. The problem with the approach is that it is hard to understand the state table and to update it. This is the approach used in many other voice mail systems and in Amanda@Work.Group.
2. Stack based procedure calling. This approach more naturally follows the flow and structure of the menus being presented. Its main problem is that it is hard to get from one place in the tree to a completely different place in the tree without using exceptions (you have to pop the stack and reload it with different procedures corresponding to the new place in the menu hierarchy). Fortunately, we don't do this very much.

Figure 1.2 shows the general structure of the menu system and the Tcl stack at each point in time.

Errors in most Amanda primitive functions are handled differently than is normally done in Tcl. Normally, when an error occurs in Tcl, an exception is raised, the message is returned in the "result" and `errorCode` is set to the error code of the error. To detect errors, you have to use catch and check the `errorCode` to see what the error was.

This is troublesome in Amanda because the people writing the Tcl code may not be programmers and won't understand exceptions. Secondly, it makes the code in the functions above unclean because you have to handle two different cases, those where the function succeeded (`catch` returned 0) and those where it failed (`catch` returned 1) quite differently. It would be better to simply return sentinel values for error codes that don't match any valid successful return value, then you could just use a big switch statement. Lastly, exceptions pop the stack. Most of the time we don't want to pop the stack. Error code returns are appropriate in this case.

The rule is this: if you think the user of this function is going to want to stay in the same procedure and take some action on an error, return a sentinel value. If you think this error is serious and we need to abort out of the menu system, raise an exception. Some errors such as hangup *are* exceptions, so they are expressed that way at the Tcl level. We want to unwind the stack and abort processing on hangup. Exceptions may also be raised manually in the programmer's Tcl code if he needs to unwind and go to another part of the menu tree (not ancestrally related).

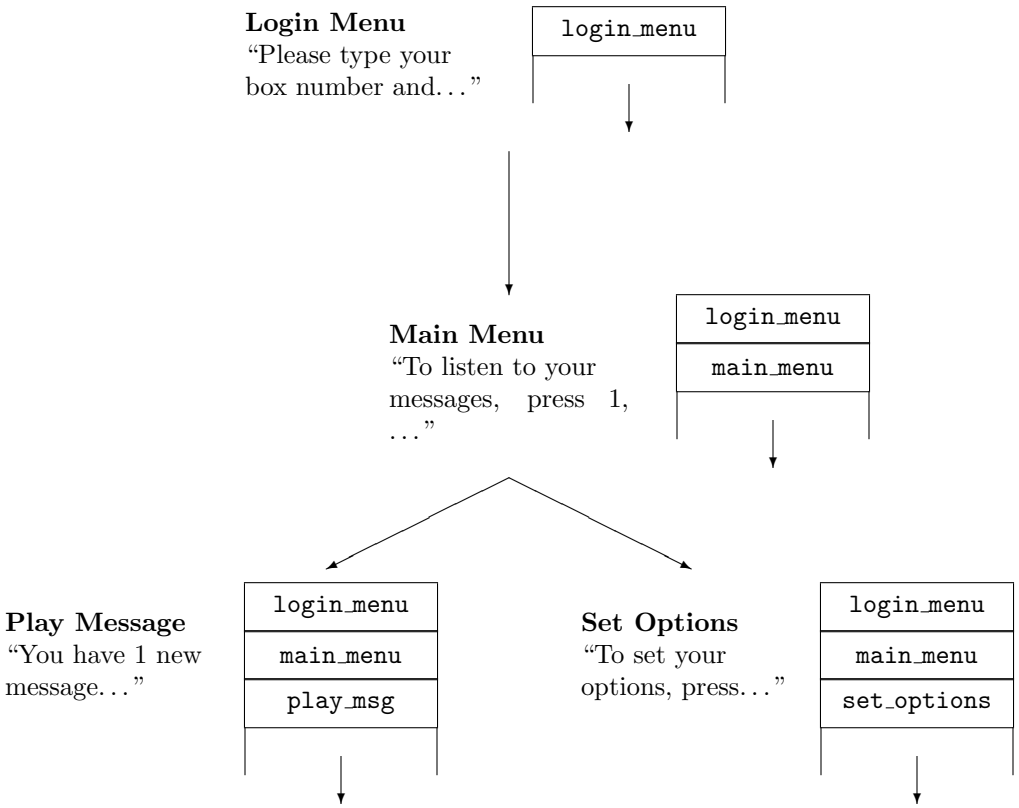

Figure 1.2: Flow control in Amanda Portal

To further make the TUI code clean, each VP device has a DTMF buffer that stores the digits that the user has typed in. Certain kernel calls such as `get_digits` will inspect this buffer before actually listening for digits and return any digits that the user has previously pressed. This separates the synchronization required between the time a user types a digit and the time the Tcl code listens for a digit. The `get_digits` call will return the digit the user typed, regardless of whether the user typed the digits before or after the `get_digits` Tcl call.

1.4 Waiting for Events

In Amanda Portal, it is often useful to start off one or more threads asynchronously and then do other things while waiting for the threads to finish. If you start off more than one thread, you may wish to wait and be notified when a thread either finishes or needs to notify you about an event that occurred. You also need a way to identify the different threads. This is done through "handles." Handles are variables that point to the name of an internal function that implements the behavior of the object the handle refers to (see section 1.2.4). These handles serve a dual purpose: they have member functions that you can execute to query and modify the objects the handles refer to and they can be used as synchronization mechanisms for threads associated with the handle. If a different thread is associated with a handle, the thread is either started when the handle is created or by a member function on the handle. When you delete a handle, your "ownership" of that object ceases and the object referred to by the handle is either freed or released for other users to use. Any threads associated with that handle are killed too.

Currently the only two type of handles that can start off asynchronous threads are VP device handles and call queueing handles. VP device handles wait until an event like playing a message finishing or the user hanging up. The queue handles wait for events like a call changing its position in the queue or a call reaching the top of the queue. Each type of handle that allows you to fork off asynchronous threads must have a `stop` member function that "stops" the operation in progress, whatever that means. In the case of VP device, it will stop the play in progress; in the case of call queueing handles, it is a no-op. Stopping is necessary because you often want to stop all the other threads from running when an event occurs. The basic wait logic looks like

Wait on various handles
After an event, foreach handle that event didn't occur on {
    Stop the handle by issuing a stop command.
}

Fortunately, it is easier to stop the various other handles than by running this loop each time. The `wait` command by default stops the other handles when it returns.

Here is the definition of the `wait` command:

---

* `wait` [`-nostop`] [`-timeout` *milliseconds*] [*handle_func*]...

Description

This is the main synchronization function for threads. Threads are referenced by handles and the functions associated with handles are passed into this command. Normally, once an event occurs and `wait` returns, all the handles are stopped (that is, "*handle_func* `stop`" is executed automatically on all the other handles). You can specify `-nostop` if you don't want `wait` to stop any of the other handles. You can then stop them individually if you wish. The `-timeout` option is used to return after a specified time. The `-all` option waits for all the *handle_func*s to return.

Return values:

`TIMEOUT` — A timeout occurred. Only returned if `-timeout` is given.

A list of 2-tuples. — The first part of each 2-tuple is the function handle name and the second part is the return result from the thread. Normally the second part will also be its own list of the form

reason args...

Only one 2-tuple is returned unless you give the `-all` option. In this case, more than one 2-tuple can be returned.

Error codes:

`CMDNOTEXIST` *handle_func*
`CMDNOTHANDLE` *handle_func*
`NOTHREAD` *handle_func*

1.5 Loadable DLLs

Amanda Portal comes with a small core which defines fundamental behavior such as the security model, the model for boxes and messages, *etc.* Additionally functionality is loaded as DLLs. DLLs have a standard interface that they must support when they are loaded into the Amanda Portal system. Loading external DLLs leaves the system open to extendibility and modularizes components.

The list of DLLs that are loaded when Amanda Portal starts up is listed in the Configuration Database (the `dlls` setting). When a DLL starts up, the `DLLMain` procedure initializes any information that the DLL may need. For example, it may interrogate the Dialogic boards in the system to see how many resources they have and then register these resources with the resource manager.

Some DLLs perform all their functionality in C (*e.g.*, the SMTP DLL) while (most) others spawn a Tcl interpreter and a new thread of control when an event happens (such as a call comes in). Privileges are associated with an interpreter (through `Tcl_SetAssocData`), so DLLs that don't spawn a Tcl interpreter cannot log in to a box and cannot have privileges associated with them.

Certain function names in a DLL are recognized as being special and are called at certain times by the Amanda Portal system. For example, when you log in to a box, a special function is called in the DLL (`interp_login`). This function can inject new privileged commands into the interpreter that only logged in users can use (the injection model). Also, on logout, another function is called that removes the privileged commands that were added on login (`interp_logout`).

The idea is to provide a set of commands for unauthenticated users that is a subset of the commands for authenticated users. The Internet DLL provides good example of this. You can `telnet` into the Amanda System and get a Tcl prompt. You have a limited set of commands you can use until you log in. When you log in, you can do more privileged things like grabbing resources (such as a voice or fax device).

You can view this set of DLLs as a forest of different C code threads listening for events and spawning interpreters when events happen. The TUI is loaded and started by the core. Each incoming call spawns a separate Tcl interpreter so multiple instances of the TUI will be running at once. You may call different entry points in the Tcl code if you want different behavior in the TUI. For example, you may have the rule that if calls come in on lines 1–3, we call the Tcl function `amanda_voicemail` and if the call comes in on lines 4–5 we call the Tcl function `amanda_forward`.

1.6 Resource Manager

The resource manager is a central registry where DLLs register the resources they provide and how many of them they provide. The resources are typed. There are port resources (LS devices), voice processing devices (VP devices), Fax devices, *etc.* When an interpreter is created and a new thread is started by a DLL, the thread may have to acquire some resources to perform its "task." To acquire these resources, the task checks the resources out of the resource manager and checks them back in when it is done. If the interpreter dies for some reason, the resources are automatically checked back in because the Tcl variable the resource is associated with gets deleted and its deletion callback fires.

When acquiring resources, it is important to be able to acquire all the resources you need atomically to avoid deadlock. For example, suppose there are only two LS resources and there are two tasks that need both LS resources. If task 1 acquires the first LS resource and then task 2 acquires the second LS resource, they both will not be able to continue until one of them releases their resource. To avoid this, Amanda allows you to allocate all the resources you need atomically.

Most resources are indistinguishable. One fax resource is interchangable with another. Telephone ports, on the other hand, are not. Furthermore, an administrator may wish to designate only certain ports to be used for certain operations, such as notifications. For this reason, *port groups* exist. When you request a port from the resource manager, you may specify that it must be a member of a given group, such as the `notify` group. These group names are not known to the system; they are created on an as-needed basis simply by assigning the ports to the groups in the configuration database.

A common scenario for a task is to acquire a single LS resource and single VP resource. For example, a call may come in and the LS resource is allocated for the port the call came in on and a VP resource is allocated to listen to the caller or the buttons the caller presses on his telephone. The caller can then possibly leave voice mail. If the call needs to be forwarded, another LS device could be allocated to send the call out on another phone line.

The resource functions are defined as follows:

---

* `list_resource_types`

Description

Lists the resource types.

Return values:
A Tcl list of the different kinds of resource types.

---

* `get_resource_stats` *resource_type*

Description

This function returns information on resource allocation. The array returned contains the following indices:

`total_defined`	Total resources of this type that exist.
`current_available`	Total resources of this type that are currently available.
`max_ever_used`	Maximum resources of this type that have ever been used at once. (This is not the maximum number of resources that have ever been used at different times. That is, suppose you have three resources, LS1, LS2 and LS3. All three LS resources have been used at one point in time, but only one resource has been in use at once.) This value is useful to see how close you are getting to saturation of a certain resource type.
`times_denied`	This value is also used to monitor saturation. The return value is another a-list. One index into the a-list is `all`, which indicates how many times a resource allocation of the resource type failed. When allocating port-type devices, you can specify an individual port or group you wish to allocate from. Denials for these ports or groups are tallied separately. The indices are the port numbers or group names. For non port-type devices, only the `all` index is set. The number of units asked for does not effect the denial number; that is, even if the process asks for 5 devices and is denied, the denial count gets incremented by 1.
`current_wait`	This value returns the number of resource units being waited on. Again, the return value is an a-list as with `times_denied`. However, the number of units asked for does effect the count returned. For example, if there is one person waiting on a type and he is waiting for 3 units of that type, 3 is returned for that type.
`max_ever_wait`	This is the maximum number of units that have ever been waited for at once. That is, it is the maximum value `current_wait` has ever been. Again, the value is an a-list like `times_denied` and `current_wait`.

Return values:
A list of the statistics for the resource type given. The list is returned as an a-list as described above.

Error codes:
`INVALIDRESTYPE` *type*

`resources_with_group` *group_name*

Description

This function returns a (possibly empty) list of all resource *types* which have one or more units in group *group_name.* This function can be used when you are writing code which you know wants to use, say, a notify port, and you don't want to care whether the system is installed with T1 or LoopStart lines. The output of this command can then be used in arguments to the `grab` and `grab_res` functions.

Return values:
A Tcl list of all resource types which have one or more units in group_name.

`grab_res` [`-timeout` *tmo*] [`-unit` *unit*] [`-port` *port*] [`-group` *group*] [`-specific_type` *specific_type*] [`-min` *min*] [`-max` *max*] [`-ascending`] [`-descending`] *type var*

Description

This function attempts to allocate a resource of type *type*. If successful, the selected unit's control function will be installed in the interpreter and assigned to *var*. The value of *type* can be any of the values returned by the `list_resource_types` or the `resources_with_group` function.

If `-group` is specified, then the returned unit must be in the named group (this is used only to select ports by logical group names rather than by number). Otherwise, you may limit the unit numbers by specifying `-min` and `-max`, or as a shorthand, `-unit` (which just sets *min* = *max* = *unit*). Using `-port` is another variation of `-group`; it will only return resources with that same port number associated with them, which will normally identify a specific port-type resource within the system.

If `-specific_type` is specified, then the returned resource must be of that specific type. This option is useful when requesting a port resource, if you need to specify exactly what type of port, loop start (ls) or T-1 (t1), you require. If `-specific_type` is not specified, then when requesting a port, any type of port which meets the other criteria will be returned.

Within the group/range specified, if more than one item is available for immediate selection, then the one with the lowest unit number will be returned if `-ascending` is specified, the highest one available if `-decending` is specified, and otherwise the least recently used one. This gives you great flexibility in how ports are used to place outbound calls. For indistinguishable objects such as fax resources, use the LRU method since it takes the least amount of time to perform the allocation because it doesn't have to search all the available units to see which one has the highest or lowest unit number (the list of units is maintained in LRU fashion, so the LRU case simply takes the first available off the list).

If `-timeout` is not specified, then `grab_res` will wait infinitely until a qualified unit (one in the group or range specified) is available. You may specify any non-negative integer for *tmo,* including 0. A value of 0 effects a poll—if a qualified unit is available, it's returned, or else no unit is selected.

When allocating items of more than one type, take care to allocate them always in the same order. Otherwise, deadlocks can occur. Also, beware of requesting a group, specific type, port, and/or unit number which does not exist in the system. Doing so will block forever, unless `-timeout` was also specified.

Return values:
If `grab_res` *fails to allocate a resource as requested, it returns* `-1`*. Otherwise, it returns the number of the unit selected.*

Error codes:
`INVALIDRESTYPE` *type*
`NOTNONNEG` *tmo*

`grab` [`-group` *group*] [`-unit` *unit*] [`-port` *port*] [`-type` *type*] [`-specific_type` *specific_type*] [`-net` *net*] [`-vp` *vp*] [`-ascending`]

Description

The `grab` command is a convenience function which allocates a network device from of type *type.* The default is to allocate from type `port` in descending order.

The device handle (Tcl variable) will be called *net,* which defaults to `net`. A DSP resource will then be attached to *net,* called *vp* (defaulting to `vp`). Resources will be allocated as in `grab_res -decending` order unless the `-ascending` option is used. The `-group` argument is the same as for `grab_res`; you may use it or `-unit`, but not both simultaneously. Similarly, `-specific_type` and `-port` are the same as for `grab_res` and have the same restrictions on its use as with that function.

If *net* and *vp* are defaulted, then the function also performs a `make_local net vp` command to make those handles' member functions be "locally" available in the interpreter.

Return values:
A return of `-1` *indicates that the network device type type was unavailable, while* `-2` *indicates that the subsequent* `dsp_attach` *failed. Otherwise, an empty string is returned to indicate success.*

Chapter 2

Security Model

Amanda Portal has a two-dimensional security model. One dimension is "who" or the identity of the user and the second dimension is "what" or what they can do. (Other systems often have a third dimension "where"). The "who" dimension is the box number. When you log in, you are assigned that number. The "what" dimension is your privileges.

2.1 Ancestors and Descendents

Boxes are related to each other ancestrally. When a box creates another box, it is the ancestor of the box and the created box is a descendent. Box creation occurs in box creation trees. The structure of these trees is maintained internally by the system and the tree structure cannot be violated. That is, if you want to delete a box, it must be a leaf node; you cannot delete internal nodes of the tree until all their descendents are deleted.

Boxes start off with a set of privileges based on those of the box they are cloned from, intersected with the set of privileges that the creating box has. You can later assign any privileges you have to a box, but you cannot assign any privileges you don't have to another box.

The topmost box of each tree is special. They are known as "superboxes" and they automatically get all privileges. Currently there is only one tree hierarchy with box 999 as the root, but in the future there may be a forest of these trees. Each topmost box in the forest will have all privileges but will only be able to control the boxes in the tree under it.

2.2 Privileges

The "what" dimension of the Amanda authentication system is handled by privileges. Each box is assigned a set of privileges that that box has. A user of a box can grant those privileges to descendent boxes, but only the privileges that the box owns. To acquire a new privilege, a proper ancestor must give it to you. The privileges are mutually exclusive and not hierarchical; that is, owning a certain privilege does not imply the ownership of any other privilege.

The top-most box can assign any privilege to any other box. Thus, the top-most box can extend the privileges in the system by making up new ones. Child boxes can then assign those privileges to other boxes, *ad infinitum*. However, the system only has built in knowledge of the privileges it is shipped with. The interpretation of any user-defined privileges is up to the Tcl code the end-user writes.

There is an interesting anomaly with privileges: you can't revoke your own privileges. You can only modify the privileges of your proper descendents and since you are not a proper descendent, you cannot create or revoke your own privileges. Your parent must do it.

Privileges are tested for using the `has_privilege` command. When you are not logged in, you have no privileges. When you log in to a box, you get the privileges associated with that box. Certain commands are associated with privileges and you will not get these commands "injected" into your interpreter unless you have the privilege. For example, if you don't have the EDIT_PMETHOD privilege, you won't get the `store_proc` command.

Because privilege state is internal, there is no way to give yourself more privileges than you already have. You have to modify privileges through the calls built into Amanda and these calls only let you manipulate privileges in predefined ways.

Privileges are actually stored as internal state attached to an interpreter with `Tcl_SetAssocData`. Therefore, because `Tcl_SetAssocData` is used, privileges can actually have values. This currently isn't used but may be in the future. Today, we only test for privilege existence.

The different privileges are

`ANNOUNCEMENTS`	This privilege allows you to create and modify announcements in your box.
`BOX_CREATION`	This privilege allows you to create non-guest boxes.
`CALL_SCREEN`	Allows you to set the call screening (`CALL_SCREEN`) and modified call screening (`MOD_CALL_SCREEN`) setting on a box.
`CHANGE_METHOD`	This privilege allows you to change the `CALLER_CODE`, `USER_CODE`, `DONE_CODE`, `RNA_CODE`, `BUSY_CODE` or `MENU*_CODE` methods for a box.
`CHANGE_TMAPPING`	This privilege allows you to change entries in the a trie mapping database (see page 129).
`CONFIGURATION`	You can change the Configuration Database settings.
`COPY_TO`	This privilege allows you to change the list of boxes another box's messages are copied to. You change this list using the "list mapping" functions in section 10.1.
`CREATE_QUEUE`	This privilege allows you to create a queue for your box or any descendent box using the `create_queue` call. It also allows you to set queue parameters with `set_queue_setting`.
`CREATE_TMAPPING`	This privilege allows you to create and destroy trie mapping database.
`CTRIGGER`	This privilege allows you to change and add notify records ("conditional triggers") to boxes.
`CTRIGGER_CODE`	This privilege allows you to create and modify the bodies of notify templates ("conditional triggers").
`DND`	Allows you to set the DND setting on a box.
`EDIT_PMETHOD`	This privilege allows you to create or change one of the persistent methods.
`EXTENSION`	Allows you to set the extension (`EXTENSION`) or location (`LOCATION`) on a box.
`GENERAL_ANNOUNCE`	This privilege allows you to create and modify announcements in the announcment mailbox.
`GREETING`	This privilege allows you to change your greeting, to set a maximum length on a greeting (`GREET_LENGTH`), or to set the delay after saying a greeting (`DELAY`).

KILL	This privilege allows you to kill all the threads of a specific user or an individual thread running on behalf of a user.
MESSAGE_MOVE	This privilege allows you to move messages from one box to another (move_msgs_to_box).
MONITOR	This privilege allows you to get status updates of tracing, ports, resources, and system.
NOTIFY_ANY_BOX	This privilege allows you to schedule a notify within another mailbox.
RESET_PORT	This privilege allows you to reset a port on a board.
RECORD_MSGS	This privilege allows you to change whether a box receives messages (RECORD_MSGS) or not and to change the maximum length of the received messages (MAX_MSG_LEN).
RESET_STATS	This privilege allows you to reset the statistics gathering counters for a box (the reset_box_stats call).
SCHEDULE_LOCK	You can create/modify autoschedule records ("time triggers")—see page 138).
SHUTDOWN	This privilege allows you to use the shutdown command, described on page 197.

The privilege commands are as follows

---

has_privilege [-box *box*] [-use_db] *privilege*

Description

This function is a predicate to determine whether your box has a certain privilege. If -box is given, test the given box instead. You can only test the privileges of your box or descendents of your box. If you are testing the privileges of the box you are logged in under, then the test is done against the state associated with the interpreter (*i.e.*, Tcl_SetAssocData). If you test another box's privileges, then the test occurs against the privilege database. You can test against the database even when testing your own box by using the -use_db flag.

Return values:
0 *or* 1

Error codes:
NOTNONNEG *box*
BOXNOTEXIST *box*
PERMDENIED *box*

---

set_privilege_value *box privilege* [*value*]

Description

This function sets a privilege on a box to the value given or the empty string if no value is given. You may not set the value for your own privilege; you must give a descendent box. Also, you can only set privileges that you yourself own. This function only sets the state in the database. Any existing interpreters logged into the box set will not see the changes. The values currently are mostly unused. Only the privilege's existence is checked for. Therefore, the default empty string is a reasonable default.

As mentioned earlier, the superbox is allowed to set any privilege for any box, including itself.

Return values:
Value privilege was set to.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`INVALIDVALUE` *value*

`get_privilege_value` [`-box` *box*] [`-use_db`] [*privilege*]...

Description

This function returns the value of the privileges for the given box. You can only inspect your privileges or descendent box privileges. If you inspect the privileges of your own box, the state associated with the interpreter is used. If you inspect the privileges of another box, the privileges are looked up in the privilege database. You can retrieve against the database with your own box by using the `-use_db` flag. The only time the values would be different is if another thread modified you privileges from under you.

Return values:
An a-list with the privilege as the index.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`PRIVNOTEXIST` *privilege*
`INVALIDVALUE` *value*

`delete_privilege` *box* [*privilege*]...

Description

This function deletes the privileges associated with the box given. You may not delete your own privileges. You may only delete the privileges of proper descendents. The root box has all privileges implicitly and cannot revoke any of its privileges.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`PRIVNOTEXIST` *privilege*

errorCode
`INVALIDVALUE` *value*

`get_privilege_keys` [`-box` *box*] [`-use_db`]

Description

This function lists all the privileges for the current box or the box given. If you retrieve the privileges of your own box, the state associated with the interpreter is used. If you retrieve the privileges of another box, the privileges are looked up in the privilege database. You can retrieve privileges from the database even when specifying your own box by using the `-use_db` flag.

Return values:
A Tcl list.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`INVALIDVALUE` *value*

2.3 Login and Logout

When you log into a box, you get the box's privileges and new commands may be injected into your interpreter that give you additional functionality. In addition, your password and your box number are set in the global array `taaPriv`. When you log out, your password and box id are removed from the global variable `taaPriv`.

The login related commands are defined as follows:

---

* `login` *box password*

Description

This function logs you into a box. Many new commands are added once you log in. If the Configuration Database parameter `max_login_attempts` is defined, then after that many failed login attempts in a row, your interpreter will be killed. This command only exists when you are *not* logged in.

Return values:
Empty string.

Error codes:
`LOGINFAILED`

`logout`

Description

Logout of a box. Messages may be expunged if `EXPUNGE_LOGOUT` is set for the box. This command exists only if you are logged in.

Return values:
Empty string.

---

* `verify_sac` *box password*

Description

This function verifies the *password* is the correct password for *box*.

Return values:
`0` *or* `1`

Error codes:
`KILLED`

Chapter 3

How Amanda Portal Interacts with Telephone Switches

Commonly, Amanda Portal is used "behind" a telephone switch, acting as a voice mail system and/or an autoattendant. Figure 3.1 shows Amanda hooked up to a phone switch. A number of outside lines come into the phone switch. Amanda is hooked up to a number of phone switch lines and may be handling more than one call at a time on these different lines. Phones are also connected to the switch. In figure 3.1 these are referenced by the users "Tim" and "Scott." Each connection on the switch is known as an "extension." Each connection on the back of the Amanda system is known as a "port." Notice that the numbers need not be the same. There is a mapping in the Configuration Database that tells Amanda which extension is hooked up to which port. There is also a mapping in Amanda that tells which extension on the phone switch is associated with which voice mail box in Amanda (in addition to other info such as the message light status, phone set name, *etc*). Lastly, there may be a serial line connected from the phone switch to Amanda to give Amanda information on calls being transferred to it if the switch uses "serial integration" (explained later).

Switches come with a variety of "smarts" and each phone switch has its own peculiar way of doing things (*i.e.*, there is no standard). Smart switches generally cost more than dumb switches but both types of hardware are extremely reliable.

Every phone switch can be told to transfer calls from one extension to another and every phone switch has an audio jack for "hold" music where you can hook up a CD player or radio to play music to the person when the person is put on hold. The switch automatically plays the music from this jack whenever a call is put on hold. If you don't hook anything up, the caller will hear silence or periodic beeps.

Integration information is information from the phone switch to Amanda that tells Amanda why the call was transferred to Amanda, which extension the call is or was destined to go to, and possibly which extension the call is coming from. Integration comes in two forms: "in-band integration" and "out-of-band integration." With in-band integration, Amanda is told information about a call by a sequence of DTMF tones on the line the call came into Amanda on. Out-of-band integration, or "serial integration" (since the data comes over a serial line), gives Amanda the call information through a separate serial line. This information says something like "The call that just came in on extension 6 was destined for extension 8 and was from extension 3." Amanda has to associate this information with the call on extension 6. Many switches send the out-of-band integration information over the serial line according to a specification called SMDI from Bellcore.

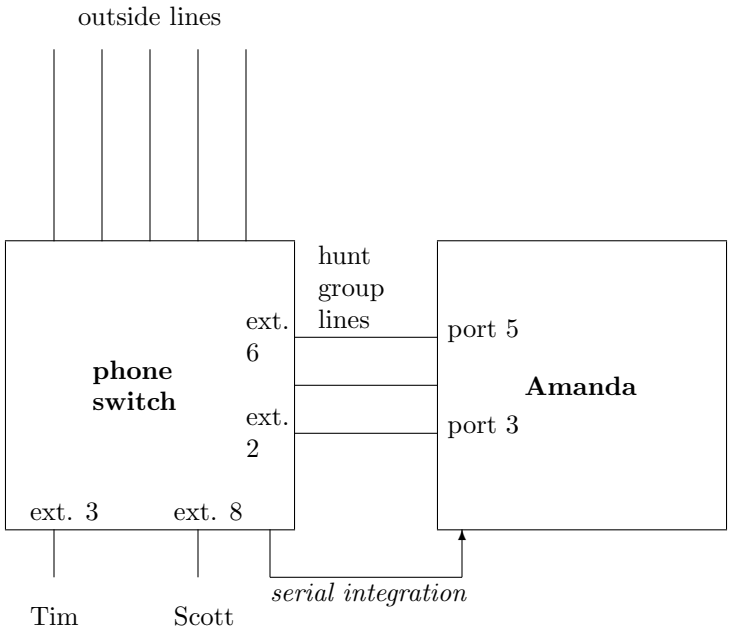

Figure 3.1: How Amanda and the Phone Switch are Connected.

With in-band integration, switches have settings in them that tell the switch which lines to send DTMF integration information on and which lines not to. For example, in Figure 3.1, the lines going to Amanda have "integration information on" and the other lines have "integration information off." If the non-Amanda lines didn't have "integration information off," then when you picked up the phone to answer a call you would hear the integration information from the switch.

Some switches don't send any integration information. With these types of switches, Amanda can't be as intelligent. When a call comes into Amanda, it can't differentiate whether a call was destined for a user's phone and the user wasn't there or whether a person just dialed in from the outside and should receive the company greeting. In the first case, Amanda should give the voice mail greeting for the box the call was destined for. In the second case, Amanda should begin call processing by playing a general greeting. Without integration information, Amanda must always give the general greeting.

If the switch provides integration information to the voice mail system, then the phone switch allows an autoattendant to do "blind-transfers." With blind-transfers, Amanda can simply transfer a call to another extension and drop the line. Once the line is dropped, Amanda can use it for other calls, such as a new incoming call from the outside. If the person does not answer the phone or the extension is busy, the switch automatically routes the call back to Amanda's hunt group[1] and integration information tells Amanda that the call was destined for the extension but didn't answer or was busy.

[1]The switch is set to call or not call Amanda back on a per extension basis according to settings in the Configuration Database. The switch is configured according to these settings when Amanda starts up.

When blind transfers are not possible, Amanda must to a "supervised transfer." With a supervised transfer, Amanda must keep the line open and listen if the person picks up the line or is busy. The port cannot be used for other purposes such as incoming calls during this time, so this form of transfer is less desirable. There are two types of supervised transfer:

1. The switch tells whether the person answered or the phone is busy through in-band DTMF. In this scenario, the switch gives different DTMF tones whether the person answered or whether the extension is busy. Amanda listens for these tones.
2. Without such integration information, Amanda must do "Programmed Call Progress Monitoring" or PCPM. With PCPM, Amanda actually listens on the phone line for the ringing, busy signal, or the human voice to determine whether someone picked up the phone or not. With these types of switches, the caller may miss the first part of the speech of the callee. The reason is that there is a delay in recognizing the human voice and transferring the call to the callee's extension. While the callee is talking, Amanda is busy transferring the caller to that extension and the caller misses the first part of the callee's speech.

The lines running to Amanda from the phone switch are not special in any way. Amanda has the ability to transfer a call to another extension only because the call was sent to Amanda. Amanda has no idea what other calls are currently going on in the phone switch, the state of the extensions, etc.

Of course, Amanda need not be used as an autoattendant. The system can be configured to take voice mail for each mailbox without trying to transfer the caller to the user's extension first. Also, since the voice processing boards in Amanda are capable of performing switching themselves over an internal TDM bus, the Amanda system can act as a PBX itself by installing the proper boards and writing appropriate driver and TUI code.

Chapter 4

Mailboxes

Each user in Amanda has one or more boxes (usually one). This box stores a variety of messages and each message can have a number of MMOs in it. The messages in boxes are arranged in folders (see page 70). Usually the messages in a box consist of a single voice MMO (*i.e.*, a single voice mail message from someone). You must login to a box to listen to the messages in it. Each box has a password so other people can't listen to your messages if they don't have the password.

There are two types of boxes: guest boxes and non-guest boxes. When you create a guest box, the system picks the box number for you. You may not select the box number to create. This is useful for something like a law office where each lawyer may create boxes for his clients so that he can leave them confidential messages for later pickup. Each client would have a different password on their box. Which box number is selected is not important. When creating non-guest boxes, the creator gets to select the box number. There are separate privileges for creating guest and non-guest boxes. To create non-guest boxes, you need the BOX_CREATION privilege. To create guest boxes, you simply need a positive `BOXES_LEFT` setting on your box. Every time you create a new non-guest or guest box, this number is decremented. So even if you have the BOX_CREATION privilege, you still must have a positive `BOXES_LEFT` setting to create a non-guest box.

When you create a box, you clone the box from another box. Cloning copies the notify and autoschedule records, the copy_to lmapping mailing list, and the basic settings on the box. When you create a box, you can specify which box to clone from. If you do not specify a box to clone from, a default box is used if it exists. There are two default boxes in the Configuration Database: one for guest boxes (`guest_defaults`) and one for non-guest boxes (`defaults_box`). These entries list the box number to clone from for the appropriate box type. Having two different default "box clone" entries is important because guest boxes may not even have a phone associated with them and non-guest boxes usually always will.

When a box is created, the password assigned depends on a number of factors. If there is no default box for the box type or if its password is blank, the password assigned is the box number; otherwise, the password is set to the same password as the default box. The intention here is to set the box password to something known if there is no password, so the user gets a reasonable default. However, we don't want people from the outside breaking into the system because they know that the box password is set by default to the box number on creation. To get around this, we allow the Amanda system administrator to set the default password in the clone box to something he knows. This prevents outside users from breaking into the system. Note that this behavior is the current behavior in Amanda@Work.Group.

4.1 Box Manipulation Functions

---

* `is_box` *box*

Description

This function is a predicate to determine whether a box exists.

Return values:
`0` *or* `1`

`create_box` [`-clone` *box*] [`-box` *box*]

Description

This function creates a new box. If `-box` is given, then a non-guest box is created with the appropriate box number; otherwise, a guest box is created. You will need the BOX_CREATION privilege to create boxes using the box number. You will need a positive `BOXES_LEFT` box setting to create guest boxes. Every time you create a guest box, this number is automatically decremented by the system. Also, every time you give a child the ability to create guest boxes, your `BOXES_LEFT` count is decremented. That is, you cannot create more guest boxes than you are allocated by giving more boxes to your child boxes. When you give boxes to your children, you do it by adjusting the child's `MAX_BOXES` setting. This value is intimately tied with the `BOXES_LEFT` setting but you cannot set the `BOXES_LEFT` setting. It is maintained by the system.

If `-clone` is given, then the box is cloned from the indicated box number. If this flag isn't given, the `defaults_box` and `guest_defaults` boxes from the configuration database are used for cloning if they exist. Autoschedule records, notify records, `copy_to` lmapping records, privileges, and box settings are cloned.

Return values:
The box number created.
GBOXLIMIT

Error codes:
`NOTNONNEG` *box*

errorCode
PERMDENIED
SYSBOXFULL
BOXEXISTS *box*
BOXNOTEXIST *clone_box*

---

delete_box [*box*]...

Description

Delete boxes. The boxes must be leaf nodes. If a box deleted is a guest box, then the immediate parent of the box has its BOXES_LEFT count incremented.

Return values:
Empty string.
BOXLOGGEDIN *delete_box*

Error codes:
NOTNONNEG *box*
PERMDENIED *login_box deletion_box*
BOXHASCHILD *delete_box child_box*
INVALIDVALUE *value*

---

reparent_box [*move_children*] *move_box new_parent*

Description

Changes the parent of a box. The box must not be currently logged in. The box and its new parent must be descendents of the logged in box making the call. The box must not currently be an ancestor of its new parent. The new parent must have enough guest allowances. This command will fail if the box has children and move_children is not specified.

Return values:
Empty string.

Error codes:
PERMDENIED *login_box move_box*
BOXNOTEXIST *move_box*
BOXHASCHILD *move_box child_box*
BOXLOGGEDIN *move_box*
GBOXLIMIT *move_box BOXES_LEFT*

---

* next_box *box*

Description

This command returns the next box (numerically) in the system after *box.* If `-1` is given for the box, return the first box.

Return values:
Box number or empty string if box given is last box.

Error codes:

errorCode	*Description*
NOTINTEGER *box*	
OUTOFRANGE *box*	User gave negative number besides -1.

---

* `previous_box` *box*

Description

This command returns the previous box allocated in the system before box. If `-1` is given for the box, return the last box.

Return values:
Box number or empty string if box given is first box.

Error codes:

errorCode	*Description*
`NOTINTEGER` *box*	
`OUTOFRANGE` *box*	User gave negative number besides -1.

---

* `get_box_children` [`-box` *box*]

Description

Returns the child boxes of the current box or the box given. If you are not logged in, you must give the `-box` option.

Return values:
A Tcl list containing the child boxes.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`BREQNLOGIN`
`INVALIDVALUE` *value*

---

* `get_root` [`-box` *box*]

Description

Returns the top most box in the box tree containing the current box or the box given. There may be a forest of box trees. If you are not logged in, you must give the `-box` option.

Return values:
The top most box number.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`BREQNLOGIN`

`set_box_password` [`-box` *box*] *password*

Description

Sets the password for the current box or the *box* given. You can only set the password for your box or one of your descendent boxes. There is no way to inspect the password of a box. You must reset the password if you forget it.

Return values:
Empty string.

Error codes:

errorCode	*Description*
`NOTNONNEG` *box*	
`BOXNOTEXIST` *box*	
`PERMDENIED` *box*	
`INVALIDPASSWORD` *password*	Password can only be numeric.

`set_box_encrypted_password` [`-box` *box*] *encrypted_password*

Description

Sets the password for the current box or the *box* given. You can only set the password for your box or one of your descendent boxes. There is no way to inspect the password of a box. You must reset the password if you forget it.

This function is exactly the same as `set_box_password` except that it takes an already-encrypted string as the password for the box, whereas set_box_password takes the plain-text version and encrypts it before storing it in the database.

Box	Key	Mutability	Deleteability	Value
1	*setting name*	read-only	deleteable	
		read-write	non-deleteable	
		ancestor-read-write		

Figure 4.1: Box Settings Attributes

Return values:
Empty string.

Error codes:

errorCode	*Description*
`NOTNONNEG` *box*	
`BOXNOTEXIST` *box*	
`PERMDENIED` *box*	

4.2 Box Settings

Each box has a number of settings that apply to it. Some of the settings are built into the system and some have special semantics when they are set. You can also add new settings to a box if you wish. If you do, the interpretation of those settings is up to the Tcl code you write. Each box setting has a number of attributes as shown in Figure 4.1.

Each setting is assigned to a box number and has a name and a value. If a setting is `read-only`, you or your ancestors can't set it. Only the superbox can create and set `read-only` settings. If a setting is `read-write`, you or any of your ancestor boxes can set it. If a setting is `ancestor-read-write`, only your proper ancestors can create or set the value. Also, settings can be `deleteable` or `non-deleteable`. If a setting is `deleteable`, then you or your ancestors can delete the value from the table. If the setting is `non-deleteable`, then it cannot be deleted from the table. The settings which are built into the system cannot be deleted, though any additional settings which were added and marked `non-deletable` can be deleted by the superbox.

The box settings functions are as follows:

---

* `get_box_setting_attrs` [`-box` *box*] [*keys*]...

Description

This function returns information about the mutability and deleteability of each of the keys given. If `-box` isn't given, get the attributes of the currently logged in box. You can read any box's attributes, including your ancestors.

Return values:
An a-list of a-lists. The first a-list is indexed by the keys given. The second a-list has two indices: `deleteability` *and* `mutability`*. Their values are the deleteability and mutability of the key.*

Error codes:
`NOTNONNEG` *box*
`KEYNOTEXIST` *box key*
`BOXNOTEXIST` *box*
`BREQNLOGIN`
`INVALIDVALUE` *value*

`set_box_setting` [`-deleteability deleteable|non-deleteable`] [`-mutability read-only | read-write | ancestor-read-write`] [`-box` *box*] [*key value*]...

Description

Set the value of box settings *atomically*. If the key doesn't exist, it is created. You can only change the settings on your box or a descendent box. Only the superbox for the tree of the box in question can set a setting to `non-deleteable` and only a superbox can delete a `non-deleteable` box setting.

The `-mutability` flag has varying restrictions depending on its value. Only the superbox can create and set a setting with the `read-only` attribute. If you wish to set a value with the `ancestor-read-write` mutability, you must set a value on one of your proper descendent boxes only. You cannot set a setting on your own box with this mutability or you wouldn't be able to modify or delete the setting. You can create a setting with `read-write` mutability on your box or any of your descendent boxes.

The `-deleteability` and `-mutability` options apply to all the key/value pairs on the command line. If you wish to issue separate attributes, you have to use different commands. The defaults are a deleteability of `deletable` and a mutability of `read-write`.

Boolean values can be given as `yes`, `true`, `1` and `no`, `false`, `0`. Integer values can be given in base 10, 8 or 16.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`KEYNOTEXIST` *key*
`PERMDENIED` *box key*

The following error codes depend on the key in question:

Error codes:

errorCode	*Description*
`INVALIDVALUE` *key value*	
`GBOXLIMIT` *limit*	For `MAX_BOXES` only.

`delete_box_setting` [`-box` *box*] [*key*]...

Description

Delete the settings from the indicated box. If the mutability is `read-only`, only the superbox can delete the setting. If the mutability is `read-write`, you or any of your ancestors can delete the setting. If the setting is `ancestor-read-write`, then only your proper ancestors can delete the setting. Only the superbox for the tree in question can delete `deleteable` settings. Of course, the deleteability attribute overrides the mutability attribute.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *key*
`KEYNOTEXIST` *box key*
`INVALIDVALUE` *value*

---

* `get_box_setting` [`-box` *box*] [*key*]...

Description

This command gets the value of the settings of keys for a particular box *atomically.* You can read any box's settings, including your ancestors'.

Return values:
Value is returned as an a-list.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`KEYNOTEXIST` *box key*
`BREQNLOGIN`
`INVALIDVALUE` *value*

---

* `get_box_setting_keys` [`-box` *box*]

Description

Get all the keys associated with a box.

Return values:
A Tcl list containing the keys for the box.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`BREQNLOGIN`
`INVALIDVALUE` *value*

Currently, the built-in box settings are as follows. Each key is non-deleteable.

Key	*Type*	*Mutability*	*Semantics*
`ABBREV_DATES`	bool	`read-write`	Controls how Amanda says the date if it is today or yesterday. If `true`, say "today" or "yesterday." If `false`, say the date. Default: `true`.
`ABBREV_GREETING`	bool	`read-write`	If `true`, say "Please leave a message at the tone." If `false`, say "Please leave a message for..." Default: `false`.
`BEGIN_REC_PROMPT`	bool	`read-write`	This setting controls the playing of the "Begin recording at the tone..." prompt for callers to this mailbox.
`BOXES_LEFT`	num	`read-only`	The number of guest boxes left for this box to allocate. If you allocate some boxes to your children, this number will be decremented. This number is decremented for guest boxes only.
`BUSY_CODE`	str	`read-write`	Code to execute when a transfer for this box detects a busy signal.
`BUSY_HOLD`	bool	`read-write`	Determines whether a caller can hold for a busy extension.
`CALLER_CODE`	str	`read-write`	Tcl code to execute when someone calls this box. Usually you will set this to a procedure name in the persistent method database. You need the CHANGE_METHOD privilege to set this value.
`CALLS_COUNT`	num	`read-only`	Number of times someone has called into this box or transferred to this box via one of the code callbacks. This value is set with the `set_internal_box_setting` call.
`CALL_SCREEN`	bool	`read-write`	Call screening boolean. If on, the caller must identify themselves to the system before the call will go through. Then when the phone rings, the system speaks the name they gave. You need the CALL_SCREEN privilege to modify this field.

Key	*Type*	*Mutability*	*Semantics*
CALLS_SECS	num	read-only	The cumulative number of seconds that the *_CODE settings are run for this box. This value is initialized with the set_internal_box_setting call and terminated with the finish_call_time call.
CALLS_TIME	num	read-only	When a caller calls into this box and one of the *_CODE settings is run, this is the time that the code started. This value is set with the set_internal_box_setting call.
COMMENT	str	ancestor-read-write	Free-form comment.
CONNECT_COUNT	num	read-only	# of times a connection has been made between caller and the person this box belongs to. The person for this box must actually pick up the phone. This count is not incremented if the caller is sent to voice mail. This value is set with the set_internal_box_setting call.
CONNECT_TIME	num	read-only	Set when a connection between a caller and the person the box belongs to is made. This value is set with the set_internal_box_setting call.
CONNECT_TONE	bool	read-write	Play a beep or don't play a beep when the called party connects to the phone.
CREATE_TIME	num	read-only	Time that this box was created in number of seconds since midnight, January 1, 1970, GMT.
CTRIGGER_COUNT	num	read-only	# of times that a notify record has been executed for this box.
CTRIGGER_TIME	num	read-only	Time the last notify record executed.
CUSTOM_BUSY	bool	read-write	Does user want his custom busy message to play? If so, the custom busy message is looked up in the MMO Lookup Table under the key busy_msg.
DATE_TIME	bool	read-write	Play the date/time when before listening to messages. Boolean.
DND	bool	read-write	Do not disturb boolean. If on, the box's greeting is never played. You need the DND privilege to change this value.
DONE_CODE	str	read-write	Code to execute when a caller has finished executing the instructions in this mailbox. You need the CHANGE_METHOD privilege to set this value. The value is Tcl code to execute and is usually a procedure name from the persistent method database.
END_REC_MENU	bool	read-write	This value controls the playing of the post-record menu for both users and callers of this box.
EXPUNGE_DAYS	num	read-write	If read messages exist for longer than these number of days, delete them.
EXPUNGE_LOGOUT	bool	read-write	Expunge the deleted messages on logout.
EXTENSION	str	read-write	Extension the phone is on. May also contain Amanda@Work.Group-style tokens. Need EXTENSION privilege to modify.

Key	*Type*	*Mutability*	*Semantics*
GREET_LENGTH	num	ancestor-read-write	Maximum number of seconds that a person's greeting can be. This is enforced in the Tcl code only; therefore, if a person uses the telnet interface, they can access the MMO Lookup Table directly and get past this restriction. This is not critical though.
CUR_GRT	num	read-write	Number of the current greeting. You must have the GREETING privilege to change this number.
DELAY	num	read-write	Delay in seconds after saying the greeting. It gives the user time to think. You need the GREETING privilege to modify this value.
ID_CALLEE	bool	read/write	When doing a supervised transfer, and we detect that the callee has answered the call, then the system announces mailbox which is transferring the call (*e.g.*, for Tech Support, for Scott Simpson, *etc.*) This allows you to have two people using the same extension, or one person acting in several capacities (sales, tech support, and accounting, for example) to answer the phone with an appropriate greeting.
LANGUAGE	str	read-write	Current language for the box. If set to a non-empty string, then the TUI will attempt to load this language whenever the user logs into this box.
LOCATION	str	read-write	Location of user. Used for call routing. May eventually store IP address or whatever. You need the EXTENSION privilege to modify this value.
LOGIN_COUNT	num	read-only	# of times user logged in.
LOGIN_DURATION	num	read-only	Time that this box has/was logged in in seconds.
LOGIN_TIME	num	read-only	Last login time.
MAX_BOXES	num	ancestor-read-write	Maximum number of guest boxes that this box can create. This value can only be modified by a proper ancestor. This value is intimately tied with the BOXES_LEFT setting. Ancestors can modify this value but they cannot change the BOXES_LEFT value. If you change this setting for a child, then your BOXES_LEFT setting is decremented appropriately. If you try to reduce MAX_BOXES for a child, you can only reduce the value by an amount less than or equal to the BOXES_LEFT value for the child.

Key	*Type*	*Mutability*	*Semantics*
MAX_MESSAGES	num	ancestor-read-write	Maximum number of messages allowed for this mailbox. If this values is 0 then an infinite number of messages can be stored. The RECORD_MSGS attribute take precedence over this attribute. You'll need the RECORD_MSGS privilege to change this value.
MAX_MSG_LEN	num	ancestor-read-write	Maximum length of message in seconds. You'll need the RECORD_MSGS privilege to change this value.
MENU0_CODE ... MENU9_CODE	str	read-write	Code to execute when a caller presses 0 through 9 while listening to this box's greeting. You need the CHANGE_METHOD privilege to change these values.
MOD_CALL_SCRN	bool	read-write	Announce name and company when doing call screening if false else announce the name and extension if true. You need the CALL_SCREEN privilege to modify this setting.
MOD_BOX	bool	read-only	The last box to modify the settings on this box.
MOD_TIME	num	read-only	Time that any settings in this box were last changed.
MSG_HI_WATER	num	read-only	Highest number of messages in box at one time.
MSGS_RECEIVED	num	read-only	Count of the number of messages that have ever been received by this box over all time.
NEW_FOLDER	num	read-write	Folder number for new non-urgent messages. (See also URGENT_FOLDER.)
PARENT	num	read-only	Parent box. Empty string for root box.
PLAY_FROM	bool	read-write	Play back who the message was from (either the box number or the name of the person).
PLAY_NEW_FIRST	bool	read-write	Play the first unheard message first else play the messages in order, regardless of whether they were heard or not.
PLAY_SKIP	num	read-write	The amount to skip forward or backwards when this user is playing a message (see the -skipby option of the play command on page 100).
RECORD_MSGS	bool	read-write	Whether this box can record messages or not. You need the RECORD_MSGS privilege to change this value.
RNA_CODE	str	read-write	Code to execute when doing a supervised transfer and a ring-no-answer condition is detected. You need the CHANGE_METHOD privilege to set this value.
RNA_RINGS	num	read-write	Ring No Answer count. Ring this many times before deciding that the extension isn't going to answer. Default: If 0, then the value of n_rings parameter in the Configuration Database; if that value is not set, then 4.

Key	*Type*	*Mutability*	*Semantics*
STAT_EPOCH	num	read-only	Time that statistical information started for this box. This value is modified by the reset_box_stats call. You must have the RESET_STATS privilege to use the reset_box_stats call.
URGENT_FOLDER	num	read-write	Folder number for new urgent messages. (See also NEW_FOLDER.)
USER_CODE	str	read-write	Tcl code to execute when user logs into their box. You need the CHANGE_METHOD privilege to set to set this value.

---

reset_box_stats [-box *box*]

Description

This command starts statistical information over for the specified box. The specified box must be either the current box or a descendent box. Also, you need the RESET_STATS privilege to execute this call.

Return values:
Empty string.

Error codes:
NOTNONNEG *box*
BOXNOTEXIST *box*
PERMDENIED

---

* set_internal_box_setting *box* CALLS|CONNECT

Description

This command sets the box settings CALLS_TIME and CALLS_COUNT if CALLS is given or the settings CONNECT_TIME and CONNECT_COUNT if CONNECT is given. This command is used in the TUI Tcl code. (This call and finish_call_time are always called when you aren't logged in.)

Return values:
Empty string.

Error codes:
NOTNONNEG *box*
BOXNOTEXIST *box*

---

* finish_call_time *box*

Description

This command tells the system that a caller who visited this mailbox is chaining to another box or has hung up; in either case, he is leaving this box, so the system should record the duration value in the `CALLS_SECS` setting. To be effective, of course, the TUI Tcl code must call this function when appropriate.

Return values:
Empty string.

Error codes:

errorCode	*Description*
`NOTNONNEG` *box*	
`BOXNOTEXIST` *box*	
`CTIMENOTSET` *box*	The call time hasn't been set through the `set_internal_box_setting` call.

Chapter 5

Multimedia Objects (MMOs)

Amanda Portal deals with a variety of media types such as voice, fax, text, *etc.* Most the objects in the system are voice messages such as prompts to play to the user when he dials up his voice mail, greetings and messages people leave in other people's boxes. However, there are also received faxes and e-mail messages in the system too. To handle all these media types, Amanda Portal has the notion of a Multimedia Object or MMO. An MMO is a handle to one of these multimedia objects. MMOs are persistent and are not deleted until the last reference to them is deleted (*i.e.*, they are reference counted). If the system dies ungracefully, Amanda Portal runs an MMO integrity check program at startup to see if there are any MMOs stored on disk that don't have any references to them. If there are, they are deleted or moved to a recovery box. To move them to a recovery box, you need to set the recovery box (`recovery_box`) and folder # (`recovery_folder`) in the Configuration Database.

MMOs are typed, are read-only or read-write, and are forwardable or non-forwardable. MMOs "know" the exact type of data that they hold; this is known as the MMO's *format.* For convenience, you can also ask an MMO what category of content it has, such as voice, fax, or text. If an MMO is read-only, you cannot change its contents because there may be other references to that same MMO; you can only dereference it yourself. When you overwrite writable MMOs, you may use a different type if you wish. If an MMO is "forwardable," then you are free to forward this MMO to another person's mailbox. If an MMO is "non-forwardable," then you cannot store it in any way, typically such as forwarding it to another mailbox. Only you can read it. The idea is that if someone sends you a message only they want you to listen to, you shouldn't be able to forward it to someone else so they can listen to it and you can't store a copy of it somewhere where someone else can access it.

MMO have "access implies listen" semantics; that is, if you can get a handle to an MMO, you can listen to it. However, you cannot get access to an MMO except through a certain set of functions provided by the Amanda system. These sets of functions specifically restrict you from getting handles to MMOs you shouldn't have handles to.

MMO handles are immutable; that is, you can get MMO handles and delete MMO handles but you cannot set them to a value. If you have a variable `v` that is an MMO handle and you try to do

```
set v 1
```

it will fail. This is implemented through the Tcl trigger mechanism which allows you to run code whenever a variable is read, written or deleted. The reason for their immutability will be discussed later. However, you can `unset` the variable and then set it to accomplish the same end result.

Every type of MMO has the following member functions:

---

* *mmo_func* `read_only`

Description

This function determines whether an MMO handle is read-only or read-write.

Return values:
`0` *or* `1`

---

* *mmo_func* `forwardable`

Description

This function determines whether an MMO handle can be forwarded to a user or not.

Return values:

`1`	MMO is forwardable.
`0`	MMO is not forwardable.

---

* *mmo_func* `ref_count`

Description

This function returns the number of references to the MMO.

Return values:
Reference count.

---

* *mmo_func* `type`

Description

This function returns the type of the MMO handle.

Return values:

`audio`	An audio MMO.
`fax`	A fax MMO.
`text`	A textual MMO.
`unknown`	The type is unknown. Until an MMO is assigned to after creation, its type is unknown. You can reassign a new value to a writable MMO and give it a different type.

---

* *mmo_func* `length`

Description

This function returns the length of the MMO. The length value returned depends on the MMO type.

Return values:

seconds	For audio MMOs, the length returned is the number of seconds of audio.
pages	For fax MMOs, the length of the fax in pages.
characters	For textual MMOs, the length of the text in characters. For Unicode, it is still the character length, which will be half the byte count since Unicode uses 16-bit characters.

Error codes:

errorCode	*Description*
`UNKNOWNTYPE`	The MMO has not been assigned to yet so the type is unknown.

---

* *mmo_func* `audio_length`

Description

This function is just like the `length` member function, but it returns the length of an audio recording in milliseconds rather than in seconds. The extra precision is sometimes needed in the TUI.

An MMO of a category other than audio will return 0 as the result of this function.

Error codes:

errorCode	*Description*
`UNKNOWNTYPE`	The MMO has not been assigned to yet so the type is unknown.

---

* *mmo_func* `format`

Description

This function returns the format of the MMO. The format returned depends on the MMO type.

Return values:

	Audio type only
`mulaw`	Mu-law.
`alaw`	A-law.
`wave`	Wave. Header specifies format.
`wave8`	Wave file known to be 8-bit mulaw.
`wave16`	Wave file known to be 16-bit linear.
`wavegsm`	Wave file known to contain audio in MS GSM format (GSM 6.10).
`gsm`	MS GSM format.
`g72x`	G.721 adpcm.
`oki-24`	Oki 24K adpcm.
`oki-32`	Oki 32K adpcm.
`sx9600`	SX9600 (I-Link proprietary) compressed audio
`basic`	Sun proprietary Audio-Basic format (Mu-law)
`rht-24`	"Rhetorex" proprietary 24K adpcm.
`rht-32`	"Rhetorex" proprietary 32K adpcm.
`linear`	16-bit PCM "Linear".
	Text type only.
`html`	HyperText Markup Language.
`unicode`	Unicode.
`ascii`	Ascii.
	Fax type only.
`tiff`	Fax TIFF/F image.
	Other
`grammar`	Compiled SAPI speech recognition grammar.

Error codes:
`UNKNOWNTYPE`

---

* *mmo_func* `date`

Description

This function returns the last-modified date on the file underlying this MMO. The date is returned as an integer number of seconds since midnight, January 1, 1970, GMT (*i.e.*, the Coordinated Universal Time).

---

* *mmo_func* `convert_to` *format*

Description

This function can be used to change the format of a writable audio MMO. It could be used, for example, just after the MMO has been recorded and before it is stored (which will convert it to non-writable). The *format* value must be one of the audio formats listed above.

---

* *mmo_func* `append_fax` *MMOs...*

Description

This function can be used to append FAX-format MMOs together. This can be useful, for example, when you have several documents that are to be sent to the same recipient in the same FAX transmission.

The *mmo_func* may be either a FAX-format MMO or it may be an empty (typeless) MMO. In the former case, the other *MMOs* will be appended onto the end of *mmo_func*. Naturally, the *MMOs* may be read-only, while *mmo_func* must be writable.

---

* *mmo_func* `path`

Description

This function returns the actual location (path) on the disk of the backing store of this MMO.

---

* *mmo_func* `date`

Description

This function returns the last-modified timestamp on the file underlying this MMO. The result is the number of seconds since 1/1/1970 GMT.

---

* *mmo_func* `id`

Description

Returns a unique identifier for each MMO within the system. The same underlying MMO will give the same unique id back regardless of how the handle to it was acquired or the name of the handle. This value can be used for checking MMO equivalence.

---

* *mmo_func* `copy_to` *format copy_from_mmo*

Description

This function copies the *copy_from_mmo* into the current mmo using the specified *format*.

Return values:
Empty string.

---

* *mmo_func* `append_to` *copy_from_mmo*

Description

This function appends the *copy_from_mmo* into the current mmo. The format of both mmos must be the same.

Return values:
Empty string.

The following calls are specific to textual MMOs. That is, they exist as member functions of textual MMOs, but not of other MMOs.

---

* *mmo_func* `get_text`

Description

This function returns the text associated with the MMO.

Return values:
Text. Unicode MMOs currently come back as strings since Tcl doesn't currently support Unicode.

---

* *mmo_func* `set_text` [`-html`] [*args*]...

Description

This function sets the value of the textual MMO to the arguments passed. If multiple arguments are passed, it concatenates the strings using spaces. Of course, the MMO itself must be writable. If `-html` is specified, then the MMO type is set to `HTML`, and otherwise it is set to `ASCII`.

Return values:
Text value MMO is set to.

Error codes:
`MMODISKFULL`

When you want to create a new MMO, you use the following function:

---

* `create_mmo` [*var*]...

Description

This function creates a new MMO. A list of variable names should be given. A new MMO is created in each variable. The MMOs created have no type. Types are assigned to them when you set the MMO. Any existing variables with the same name are deleted first.

Return values:
Empty string.

Error codes:
`MMODISKFULL`

---

* `dup_mmo` *var*

Description

Duplicate an existing MMO handle into a new variable `var`. The new variable will represent its own reference to the MMO, so calling this function will increment the MMO's reference count. It does *not* copy the contents of the MMO into a new backing store location.

Return values:
Empty string.

`print_mmo` *var printer_name*

Description

Print an existing, viewable, MMO handle using the appropiate application through shell extensions. This will use the specified *printer_name*. This function uses the `printto` shell extension. By default, *Microsoft Notepad* is used to print text and html files. Tiff files, however, do not print well with the default Microsoft viewer. We recommend using `Amanda Messenger` to print tiff files.

Return values:

`EMPTY`	The *var* has no content.
`NON_PRINTABLE`	The *var* is not viewable.
`NO_PRINTER`	Unable to get the default printer.
`NO_PRINTER_DATA`	Unable to get the printer information.
`INVALID_PRINTER`	*printer_name* is invalid.

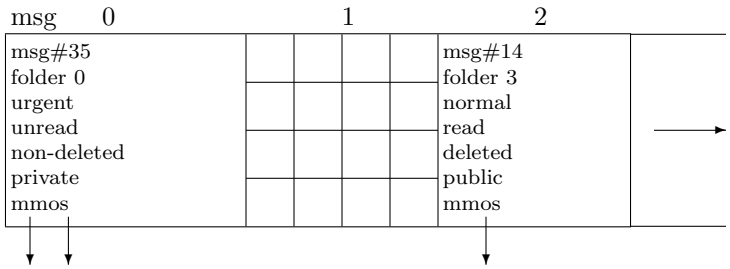

Figure 5.1: Internal Message Representation.

`PRINT_FAILURE` Failed to print the *var*.

5.1 Messages and Folders

Messages consist of zero or more MMOs and some other metadata (such as the priority, privateness, who it is from, etc.). The most common messages contain a single voice MMO; when a user calls you and you aren't in, they leave a message in your box. This message contains a single voice MMO. You can receive faxes too and these can be stored in messages. Messages with multiple MMOs usually have been forwarded to you from another user who recorded a prefix. You may receive a fax message and then decide to forward the fax message to someone else. When you do this, you decide to precede the fax message with a voice mail message that says "Jim, check out the prices for the new Gateway 2000 computer on this fax" followed by the fax. You then forward these two MMOs to somebody else as a new message. When the recipient listens to the message, Amanda announces that the current message contains a fax and then plays the voice portion. The recipient can then receive the fax over the same phone line, send it to a printer, etc., by selecting the appropriate option from the voice menu.

Some messages are private. Private messages cannot be forwarded onto other users. Messages also have a priority level which is "urgent" or "normal." This priority level is independent of the "privateness." Each message is also marked with as "read" or "unread" (*i.e.*, "heard" or "unheard" for voice messages). You mark a message as "read" by setting the `-read` flag to the `set_msg_attr` command (see page 74).

Messages are divided up into folders numbered 0 through 9. In your box settings, you specify where you want received messages to go (see `NEW_FOLDER` and `URGENT_FOLDER`). When you log in, the TUI may announce how many new normal and urgent messages you have.

Each folder also has the concept of a "current" message. As you move forward or backward in the folder, the internal current message pointer gets updated. When you switch from one folder to another, you start before the first message in the folder (*i.e.*, you must do a `get_next_msg` call to get the first message).

All the messages in all the folders are actually stored in a single file for efficiency. When a new message is added to any folder, it is simply appended to the file. The internal representation looks like Figure 5.1. Whenever message numbers are passed into or returned by Tcl calls, the internal message numbers (35 and 14 in figure 5.1) are used. You never pass in the relative message numbers (the 0, 1, 2 above) in the folder and you will never get them back from the internal functions.

Deleted messages are not deleted right away. Instead, they are marked for deletion. Later they can be undeleted by using the `set_msg_attr` command. To delete a message permanently, you can use the `expunge` command. The second box above shows a permanently deleted message. Permanently deleting a message leaves a gap in the file that may be garbage collected at a later time. When the file contains only permanently deleted messages on the end, it is truncated to a shorter length. Notice that each message is not labeled with a number within its respective folder; its location is computed by searching left to right. Also, there is only one current message pointer for all folders so when you switch folders, the current message pointer is reset to the beginning.

Message may be expunged automatically according to a setting on your box (`EXPUNGE_LOGOUT`). If this is set to `true`, deleted messages will be expunged on logout or when the interpreter is deleted (deleting an interpreter executes the `logout` function automatically).

To simulate the Amanda@Work.Group functionality using this scheme, we can simply stick all messages in folder 0 and use the normal/urgent and read/unread flags to determine the message status. For non-Amanda@Work.Group behavior, we can stick urgent messages in one folder and normal messages in another folder.

The following are the message related functions:

`change_folder` *folder*

Description

This function changes the folder to the one given. The current message is set to right before the start of the first message in the folder. That is, you must use `get_next_msg` to get the first message in the folder.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *folder*
`OUTOFRANGE` *folder*

`get_current_folder`

Description

This function returns the current folder number. When you first log in, folder 0 is the current folder.

Return values:
Current folder number.

`get_next_msg` [`-urgent`|`-normal`] [`-read`|`-unread`] [`-deleted`|`-undeleted`] [`-wrap`] *msg_var*

Description

This function returns the next message in the current folder. You can get the next message according to a variety of criteria. These criteria are specified by the `-urgent`/`-normal`, `-read`/`-unread` and `-deleted`/`-undeleted` flags. Normally, if you give no flags, then the next message is received no matter what its attributes are. If you give one of the flags, then the message found must have that attribute. Also, you cannot give both flags in the above pairs or a match would never succeed (*i.e.*, a message can't be both urgent *and* normal).

The `-wrap` option says to start over at the beginning if you don't find the message off the end of the folder. *msg_var* is a variable to be filled out with information about the message. If the variable already exists, it is deleted first. To reset the message to the beginning, set your folder to the current folder (which has the side effect of resetting the message pointer too).

Return values:
If no message is found that meets the criteria, then the string `NOMESSAGE` is returned. Otherwise, an empty string is returned and a variety of information is placed in *msg_var*. This information is basically divided into two logical sections: 1) MMO handles for each part of the message and 2) information about the message itself. For each MMO in the message, an index is set in *msg_var* that indicates the MMO's position in the message and what the internal function for the MMO is. That is, if the message contains two MMOs, then *msg_var* may get set as follows:

```
set msg_var(1) _mmo0
set msg_var(2) _mmo1
```

where `msg_var(`*n*`)` is a handle that has a deletion callback for the MMO corresponding to the internal function `_mmo0`.
The *msg_var* variable also gets filled with other information about the message. The indices are as follows:

Key	*Value*
`urgent`	`1/0`
`read`	`1/0`
`deleted`	`1/0`
`subject`	subject MMO for message. Notice that subjects can be spoken so they are MMOs, not text. This value is not set if there is no subject.
`count`	Number of MMOs in the message, excluding the subject.
`folder`	*folder #*
`private`	`1/0`
`relay_phone`	Relay phone number. This is only returned if there is a relay phone number for this message. The relay phone number is passed in to `send_msg`.
`recipient_list`	A list of box numbers and/or lists this message was addressed to.
`from`	The box number this message is from. If the message was sent by a logged out interpreter, this entry is not set.
`forwarded_by`	If the message was forwarded, who it was forwarded by. This will not be set if the message wasn't forwarded.
`receipt_message`	`read/deleted`. Only set if message is a receipt message. This message was sent in response to another box reading the message (*i.e.*, setting the state of the message to read. See `set_msg_attr`). This is set on the response message you get back, not on the message you sent. Receipt messages have no MMOs, so this flag differentiates between a regular message with no MMOs and a receipt message.
`date_time`	Date and time of message in # seconds since 1970 in GMT.
`msg_number`	Message number.
`msg_id`	This is an internally created number for the message that is unique within the Amanda system. The system generates this value.
`caller_id`	Caller id for the message. If no caller id received, it is not set.
`alternative`	This is a boolean value (0 or 1) which indicates whether the message's components are alternatives, in the MIME sense, or whether they are truely separate components. This value will be set only when parsing a multipart/alternative email message.

It is important to note that MMOs in a message are order dependent and they may be of different types. For example, if the first MMO is a voice message that says, "The following is a fax." and the next MMO is a fax, then it is important that you play the messages in order. Unfortunately, you cannot simply pass all the MMOs to the `play` command in one fell swoop if this is the case because the `play` command only plays voice messages.

`get_prev_msg` [`-urgent`|`-normal`] [`-read`|`-unread`] [`-deleted`|`-undeleted`] [`-wrap`] *msg_var*

Description

This function is the same as `get_next_msg` except it goes backward instead of forwards within a folder. If the user is positioned prior to the first message in the folder (as after `change_folder`), then `get_prev_msg` will wrap to the last message in the folder, regardless of whether the `-wrap` argument is used. But if positioned on the first message in the folder, then it will wrap to the last message only if `-wrap` is specified.

Return values:
See `get_next_msg` *for return value.*

`set_msg_attr` [`-read`|`-unread`] [`-urgent`|`-normal`] [`-deleted`|`-undeleted`] [`-subject` *mmo_func*] [*msg_num*]...

Description

This function allows you to set some attributes of a message. If you don't give any message numbers, the current message is used. If you give a message number, the internal message number is used. (Remember, relative message numbers are never used.) You can even change deleted messages before they are expunged. The first time that you change an unread message to an read message or delete an unread message by switching the state flag, a return receipt message is sent to the sender if "return receipt requested" is set. A return receipt is also sent when the message is expunged without being read.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *msg*
`MSGNOTEXIST` *msg*
`CMDNOTEXIST` *mmo_func*
`CMDNOTHANDLE` *mmo_func*
`MMOWRONGTYPE` *mmo_func*
`NOCURMSG`

`move_msg` *folder* [*msg*]...

Description

This function moves messages from the current folder to the given folder. When the messages are moved, they are inserted into the new folder in time order. That is, they may be interspersed into the new folder, depending on when the existing messages arrived in the new folder. (The messages aren't really moved. Their folder attribute is just changed.)

Return values:
Empty string.

Error codes:
`NOTNONNEG` *folder|msg*
`OUTOFRANGE` *folder*
`MSGNOTEXIST` *msg*

`expunge`

Description

This function permanently deletes all the messages marked as `deleted` in all the folders. It also sets the folder to 0 and sets the message cursor to before the first message.

Return values:
Empty string.

`get_folder_stats`

Description

This function returns information about each message in the current folder.

Return values:
A list of 2-tuples is returned. The first part of the 2-tuple is the internal message number of the message. The second part of the 2-tuple is an a-list describing information about the message. The location of the 2-tuple determines the relative message number. For example, the list returned may look like

```
46 {urgent 1 read 0 ...} 3 {urgent 0 read 1 deleted 0 ...} ...
```

Relative message 0 has internal message number 46, message 1 has internal message number 3, etc. *The information values returned are the same as those filled in by* `get_next_msg` *with the exception of MMOs being returned.*

`has_next_prev`

Description

This function returns an A-list containing two items: `next` and `prev`. Each is followed by a value specifying the number of the next/previous message relative to the current message. If there is no non-deleted next/previous message, the message number will be specified as -1.

`get_box_stats` [`-box` *box*]

Description

This function returns information about all the messages in a box, regardless of which folder they are in. It is used to get summary information about a box. If `-box` is not specified, then information about the currently-logged-in box is returned; otherwise, information about *box* is returned. When using `-box`, then *box* must be a descendent of the currently-logged-in box.

Return values:
The return value is an a-list. The entries are

Key	*Value*
`msgs`	Total # of messages in all folders.
`unread_msgs`	Total # of unread messages.
`deleted_msgs`	Total # of deleted messages.
`urgent_msgs`	Total # of urgent messages.
`normal_msgs`	Total # of normal messages.
`urgent_unread_msgs`	Total # of unread urgent messages.
`normal_unread_msgs`	Total # of unread normal messages.

`goto_msg` *msg msg_var*

Description

This function goes to a message using the message number. This function automatically switches the current message and folder corresponding to message *msg* when it is called.

Return values:
The return value is the same as `get_next_msg`.

Error codes:
`NOTNONNEG` *msg*
`MSGNOTEXIST` *msg*

`set_message_box` *box*

Description

This function is use to set which mailbox messages are being currently accessed. The default is that you can only access the messages of your own mailbox. Once set for another *box* then you just use the same tcl message commands as if they where your own message of your mailbox. To start accessing yoru own messages again, just call this function again by passing in your mailbox. For security purposes the mailbox that you want to access its message of, *box*, must grant you permission, unless it is your own mailbox. *box* maintains a list mapping named `msg_access` that lists all the mailboxes that has access rights to do anything to its messages. In order to change this list, you must be the owner mailbox of the list or have the privilege `msg_access` prepended by the mailbox that you want to change, for example *msg_access100*. If a mailbox forwards a message for the owner mailbox, the true mailbox doing the forwarding will be marked as the `forwarded_by`. If a return receipt is requested of a message that is first read by a mailbox other than the owner then the return receipt will be from the mailbox causing the return receipt, not the owner mailbox. Note that calling this function will cause a possible purge and/or expunge of the previous mailbox's messages that was being accessed, for example the first time you call this in a session then that purge or expunge will be on your own mailbox, just like if you were logging out of your mailbox.

Error codes:
`PERMDENIED` *box*

Messages can be sent either to individual boxes or mailing lists. When sending to mailing lists, the contents of the mailing list are automatically looked up in the "list mapping database" (see page 125). Recipients are specified with the syntax:

box or *list*`*`[*box*]

To send to a box, you just give the box number. To send to a mailing list, you send to a list number followed by a '`*`'. Optionally, you can also give a box number for the mailing list if the mailing list doesn't correspond to the box you are currently logged in under. In fact, you must give the box number if you are not logged in. Even users that are not logged in can send messages (using `send_msg`) and check whether a box receives messages or not.

Here are the calls related to sending a message:

`*` `check_recipients` [*box*|*mailing_list*]...

Description

This routine validates the recipients of a message. Boxes are validated to see if they exist and store messages. A box "stores messages" if either it stores messages directly and/or has its messages copied to another box. This rule is not transitive. That is, if there are three boxes A, B, and C and '→' indicates copying messages and A→B→C and A receives a message, it is stored in A (if A stores messages) and in B, but not C. Mailing lists are only validated by checking the box number (if given). We cannot check if the mailing list exists because there is no difference between a mailing list not existing and being empty. (That is, the lmapping routines return the same result for non-existence and emptiness). Also, if allowed to store messages then the limit of the number of messages is checked. If you are not logged in and execute this call, the box number for a mailing list must be given.

Return values:
The list of boxes or mailing lists that are invalid is returned. Invalid syntax arguments are returned also.

---

* `send_msg` [`-urgent`] [`-private`] [`-receipt`] [`-subject` *mmo_func*] [`-relay` *number*] [`-mmos` *mmo_func_list*] [`-at` *time*] [`-fcc` *folder*] [`-caller_id` *number*] [`-expire` *time*] [`-expire_absolute`] [*recipient*]...

Description

Send a message to a list of recipients. Recipients can be boxes or mailing lists with the syntax as described above. The options are as follows:

`-urgent`	Mark the message as urgent.
`-private`	Mark the message as private. The recipient will not be able to forward the message or any of its contents to another user.
`-receipt`	Send a return receipt. When the user reads the message, you will get a message back saying it has been read. If the user deletes the message without reading it, you will get a message back saying the message was deleted without being read. This is implemented internally in the system when the target box sets the unread or deleted flag on the message. When this is done for the first time on the message, a return receipt message with no MMOs is sent to the originating box with the `receipt_message` flag set to read or deleted.
`-subject` *subject*	Set the subject of the message to *subject*. Notice this is an MMO, so it can be voice (or a fax!).
`-relay` *number*	Set the relay field in the message to *number*. This number can be used by a notify template which is executed by one of the box's notify records. This template then can sends a page to the recipient telling him the number in the relay field so that he can return the call directly without having to listen to voice mail first.
`-mmos` *mmo_func_list*	Send the MMOs listed to the person. This option will usually always be given because you will usually be sending voice mail to people. It is possible to send messages to somebody without any MMOs though. You may wish to send just a subject for example.
`-at` *time*	Send the message at the indicated time. *time* is given in the number of seconds since 1970 and is always given in GMT. Use time manipulation functions described in section 17.2 to manipulate time values.
`-fcc` *folder*	File a copy of the message in the indicated folder.
`-caller_id` *number*	Set the `caller_id` for the message. The `caller_id` field in the message will be set when `get_next_msg` is used.
`-expire` *time*	Expire the message at the absolute *time*. *time* is given in GMT and is the number of seconds since 1970 (the standard time count). If this is not given, the message does not expire.
`-absolute`	Expire the message even if it was heard.

MMOs sent by this command will automatically be made read-only if they are not already. This prevents modification of the MMO after sending it.

Return values:
The list of recipients that the send failed to. This list is usually empty. There is a race condition between the time that you check whether a box is valid and the time you make the `send_msg` *call. If the box is deleted in the interim, it will be returned as the result of this function but the message will be sent to the other boxes.*

Error codes:
INVRECIPIENT *recipient*
INVALIDTIME *time*
NOTNONNEG *folder*
OUTOFRANGE *folder*
CMDNOTEXIST *mmo_func*
CMDNOTHANDLE *mmo_func*
UNKNOWNTYPE *mmo_func*
MMONOTFORW *mmo_func*

`forward_msg` [`-receipt`] [`-at` *time*] [`-urgent`] [`-private`] [`-subject` *subject*] [`-mmos` *mmo_func_list*] [`-fcc` *folder*] [`-expire` *time*] [`-expire_absolute`] *msg* [*recipient*]...

Description

Forwards a message to the recipients listed. The MMOs in *mmo_func_list* are prepended to the original message MMO list. See the `send_msg` function for a description of the options. The MMOs in *mmo_func_list* are made read only as described in the `send_msg` command.

Return values:
See the `send_msg` *call for the return value.*

Error codes:
INVRECIPIENT *recipient*
INVALIDTIME *time*
CMDNOTEXIST *mmo_func*
CMDNOTHANDLE *mmo_func*
UNKNOWNTYPE *subject_func*
MMONOTFORW *subject_func*
MSGNOTEXIST *msg*
MSGNOTFORW *msg*

`move_msgs_to_box` *from_box dest_box*

Description

This command moves all messages from one of your descendent boxes to a different descendent box. The messages stay in the same folder and retain their other attributes. You need the MESSAGE_MOVE privilege to execute this command. Neither box can be logged in at the time and the *dest_box* must not have any messages already.

This command is commonly used by hotels when a guest moves to a different room.

Only the `normal` notify type is fired off for the mailbox that the messages were moved to, no matter what the status (new, old) or types (normal, urgent, relay) of the messages that were moved. A `pickup` notify type is fired for the mailbox that the messages were moved from.

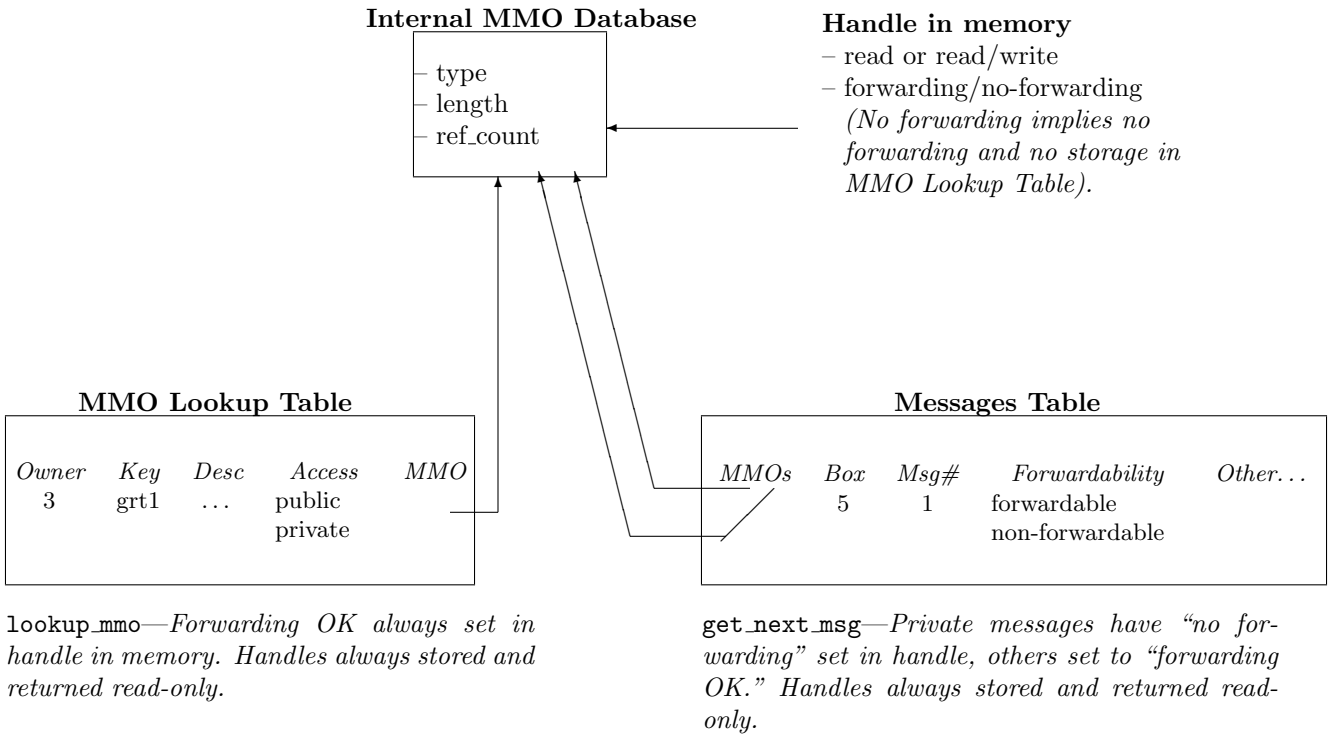

Figure 5.2: Internal MMO database structure.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *box*

5.2 MMO Database

MMOs are stored internally on disk as files. The types of these files vary depending on whether the MMO is a voice file, text file, *etc.* Also, access to MMOs is restricted by the "kernel interface" of Amanda (the calls built in). If access were not restricted, anybody could listen to anybody else's messages.

MMOs are stored in an internal database that is referenced by two other databases: the MMO Lookup Table and the Messages Table. Figure 5.2 shows the architecture. The Internal MMO Database stores the MMO information persistently (as files) and stores some meta information about the MMO such as its type, length, and a reference count.

The MMO Lookup Table is a persistent MMO storage area for MMOs. It stores such things as a person's greeting, the names they assign to folders, *etc.* Entries in this table can be public or private. Public entries can be accessed (*i.e.*, get an in memory handle) by anyone, while private entries can only be accessed by the owner of the box. In addition, MMO handles are always stored and retrieved from the MMO Lookup Table read-only. When you store an MMO handle that is read-write (*i.e.*, an MMO handle created by `create_mmo`), the MMO handle automatically converts from read-write to read-only as soon as you store it in the MMO Lookup Table. This prevents you from overwriting the MMO if it is in use. No one else can have a writable handle to the MMO because writable handles are created with `create_mmo` and this call creates interpreter specific handles. Any person can forward the MMO extracted from this table to anyone else. When the MMO handle is created by `lookup_mmo`, "forwarding OK" is always set to true.

The Messages Tables stores the MMOs that comprise an individual message in your box. You can't access this table directly except through the `get_next_msg` and `get_prev_msg` functions (see page 72). These functions will return a list of MMO handles for the current message in your box. The handles returned will all be read-only. If you then store the handles in the Lookup Table, they too will be read-only. Private messages have "no forwarding" set in the handle when they are created by `get_next_msg`. If an MMO handle has "no forwarding" set, you cannot forward this message to another user. In addition, if "no forwarding" is set, you cannot store the message in the MMO Lookup Table. If you could, then you could bypass security by extracting the message from the MMO Lookup Table which always sets forwarding to "forwarding OK." The whole forwarding concept is built into the system to support people sending you private messages. These messages shouldn't be able to be heard by anyone except the box owner the message was destined to.

Whenever you create a new MMO handle with the `create_mmo` function, the handle is always read-write and forwardable. As soon as you send the MMO to a person in a message, forward it or store it in the MMO Lookup Table, the handle is immediately made read-only to prevent you from modifying it if someone else is using it.

Recall that MMO handles are really special variables bound to internally created functions. The object-oriented functions supported by an MMO handle are rather limited (see page 64). Functions exist for things like querying MMO handles about the read-only or read-write aspect, whether the MMO handle is forwardable or not, what type the MMO handle is, *etc.* You cannot play an MMO handle using its member functions; to play the MMO handle, you have to use a VP device's member function and give the MMO handle as an argument (see Chapter 6). Also, composing a forwarded message out of any "non-forwardable" MMO handles is *verboten.*

Here are the commands for manipulating the MMO Lookup Table:

---

* `lookup_mmo` [`-box` *box*] *key var*

Description

This function looks up an MMO in the MMO Lookup Table. You can use the `-box` option to specify a descendent box to look up. Every MMO returned is read-only. If *var* exists, it is silently replaced.

Return values:

Empty string	The MMO handle was successfully filled in.
`EMPTY`	The key was found, but the MMO handle was empty. In this case, *var* is not set.
`NOTFOUND`	Key not found.

Error codes:

errorCode	*Description*
`NOTNONNEG` *box*	
`PERMDENIED` *box*	You are not allowed to access the MMO. See `lookup_mmo_attr` for access rights.
`BOXNOTEXIST` *box*	
`BREQNLOGIN`	
`INVALIDVALUE` *value*	

---

* `lookup_mmo_attr` [`-box` *box*] *key* [*field*]...

Description

Lookup information in the MMO table about everything *but* the MMO. *field* is a field to look up. The permission restrictions are the same as for `lookup_mmo`. In fact, you cannot even look up the field `private` if you don't have permission to see if the field is private. That's OK. The `lookup_mmo_keys` command doesn't return private keys when you don't have permission to look at them. The set of fields are as follows:

`description`	Description field.
`private`	Privateness access field
`box`	Owner mailbox field.
`key`	Key field.

Valid values for the private field:

`world`	Anyone can access the MMO, even non-logged in users.
`logged_in`	Any logged in mailbox can access the MMO.
`family`	Only ancestors, descendants, and owner can access the MMO.
`owner_ancestors`	Only ancestors and owner can access the MMO.
`owner_descendants`	Only descendants and owner can access the MMO.
`owner_only`	Only the owner can access the MMO.

Return values:

a-list	An a-list of the values found.
`NOTFOUND`	The *key* was not found.

Error codes:

`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`FIELDNOTEXIST` *field*
`INVALIDVALUE` *value*
`BREQNLOGIN`

errorCode

---

* `lookup_mmo_keys` [`-box` *box*] [*pattern*]

Description

Lookup all the keys in the MMO table for the current box. If `-box` is given, lookup the keys for the given box instead. In this case, the access rights if the MMO is checked. If access is granted then the key is listed. If a pattern is given, the keys returned must start with that pattern. See `lookup_mmo_attr` for access rights.

Return values:
An regular Tcl list of all the keys.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`INVALIDVALUE` *value*
`BREQNLOGIN`

`store_mmo` [`-box` *box*] [`-mmo` *mmo_func*] [`-description` *value*] [`-private` *value*] *key*

Description

Populate the MMO table. Values are stored under the given key. When modifying values in the table, any values not set are left unmodified. You can set the MMO or any of the other attributes in the table using this function. This function will also create the *key* if it doesn't exist. The defaults on a newly created *key* are an empty MMO handle, the empty string for the description and `-private world`. You may also give the empty string to unset the MMO. If the MMO you store is currently read-write, it will magically change to read-only after calling this function. See `lookup_mmo_attr` for the `private` values.

Return values:
Empty string

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`FORWNOTSTORE` *func_name*
`CMDNOTEXIST` *mmo_func*
`CMDNOTHANDLE` *mmo_func*
`INVALIDVALUE` *value*

`delete_mmo` [`-box` *box*] [*key*]...

Description

Deletes the keys from the current box or the given box. Ancestral permission relationships apply.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`KEYNOTEXIST` *key*
`PERMDENIED` *box*
`INVALIDVALUE` *value*

5.3 Announcements

Announcements are messages that are sent out to a group. Unlike messages, announcements are owned throughout its life time by the one who created the announcement. Where as in messages, once it is receive the recipient owns the message. Therefore announcements can be changed after it was created by the one who created it. Another difference between announcements and messages is the type of recipient. Message recipients are mailboxes or mailing list. Announcement recipients is just an object name, the application defines what the object name is and who the true recipients are. The last major difference between announcements and messages: is the support for different languages of the same announcement. A message is usually in one language for it is directed at a particular recipient. MMOs are used to store the different languages. For example the administrator of the system can have a Message of the Day announcement that everyone in the system listens to.

Announcements can be marked heard by a particular mailbox to remember what is heard or not. But, if the owner of the announcement changes the announcement, it is marked unheard by everyone in the system.

To be able to create and manipulate announcements you must have the `ANNOUNCEMENTS` privilege.

Announcements can be set up to be manipulated by several different mailboxes. The announcements are stored in a common mailbox designated by the global configuration parameter *announcement_box.* You must have the `GENERAL_ANNOUNCE` privilege to create and/or manipulate announcements in the *announcement_box.*

The following are the commands to manipulate announcements:

`owner_create_announcement` [`-category` *category*] [`-begin_time` *begin_time*] [`-end_time` *end_time*] [`-recip_type_list` *recip_type_list*] [`-outbound_total` *outbound_total*] [`-general` *boolean*] [`-mmo_list` *mmo_list*]

Description

The function creates a new announcement. The `category` states the general subject of the announcement. The `begin_time` and `end_time` defines when the announcement can be listened to. A recipient can not get an announcment outside of these times. The `recip_type_list` is a list if names that the application defines as to who gets the announcment. The `outbound_total` defines how many times the announcement is to be called out. The application itself handles the action of doing the outbound call. The *-general* defines if this announcement is stored in the *announcement_box*. The `mmo_list` is a list of mmo_var language pairs stating the actual announcement in each language. The first pair will become the default if the recipient's language is not available. In this case the language will be called `default.`

Return values:

id	This is the id of the newly created announcement.
`INVALIDTIMES`	The *begin_time* and/or *end_time* is invalid.
`PERMDENIED`	Not allowed to create an announcement in *announcement_box*.
`BOXNOTEXIST`	The configuration parameter *announcement_box* is invalid.
`BADMMOLIST`	The *mmo_list* is invalid, possible bad mmo_vars.

Error codes:

errorCode	*Description*
`CONFNOTEXIST`	
announcement_box	

`owner_edit_announcement` *id* [`-category` *category*] [`-begin_time` *begin_time*] [`-end_time` *end_time*] [`-recip_type_list` *recip_type_list*] [`-outbound_total` *outbound_total*] [`-mmo_list` *mmo_list*]

Description

The function edits an existing announcement. The *id* states which announcement to edit. The `category` states the general subject of the announcement. The `begin_time` and `end_time` defines when the announcement can be listened to. A recipient can not get an announcment outside of these times. The `recip_type_list` is a list if names that the application defines as to who gets the announcment. The `outbound_total` defines how many times the announcement is to be called out. The application itself handles the action of doing the outbound call. The `mmo_list` is a list of mmo_var language pairs stating the actual announcement in each language. The first pair will become the default if the recipient's language is not available. In this case the language will be called `default.`

Return values:

`INVALIDID`	The id of the announcement to edit is invalid.
`INVALIDTIMES`	The *begin_time* and/or *end_time* is invalid.
`PERMDENIED`	Not allowed to edit an announcement.
`BOXNOTEXIST`	The configuration parameter *announcement_box* is invalid.
`BADMMOLIST`	The *mmo_list* is invalid, possible bad mmo_vars.

Error codes:

errorCode	*Description*
`CONFNOTEXIST`	
announcement_box	

`owner_delete_announcement` *id*

Description

The function deletes an existing announcement. The *id* states which announcement to delete.

Return values:

`INVALIDID`	The id of the announcement to edit is invalid.
`PERMDENIED`	Not allowed to delete an announcement.
`BOXNOTEXIST`	The configuration parameter *announcement_box* is invalid.

Error codes:

errorCode	*Description*
`CONFNOTEXIST` *announcement_box*	

`get_announcement` *id var*

Description

The function gets an existing announcement. The *id* states which announcement to get. The var will be the name of the associative array that contains the relavent information of the announcement. If the owner of the announcement is doing the getting then all the attributes are returned. If a recipient is doing the getting then only the following attributes are returned: `category`, `owner`, `heard`, *mmo_vars*, and `id`. The *mmo_vars* uses the name of the language that each mmo represents.

Return values:

`INVALIDID`	The id of the announcement to edit is invalid. Possibly the `id` is good, but the it is not the right time as defined by the `begin_time` and `end_time`
`PERMDENIED`	Not allowed to get an announcement.

`owner_list_announcement` [`-general` *boolean*] [`-category` *category*]

Description

The function gets a list of existing announcement this mailbox can manipulate. The `general` defines whether or not to include general announcements. The *category* filters the list to only include those announcements of the specified category.

Return values:

id list	A list of announcement ids

Error codes:

errorCode	*Description*
`CONFNOTEXIST` *announcement_box*	
`INVALIDVALUE` *value*	

`recip_heard_announcement` *id*

Description

The function marks that this mailbox heard the announcement.

Return values:

`INVALIDID`	The id of the announcement to mark heard is invalid.

`recip_list_announcement` [`-recip_type_list` *recip_type_list*] [`-category` *category*]

Description

The function gets a list of existing announcement this mailbox can receive. The *recip_type_list* filters the announcements for only those recipient types. The *category* filters the list to only include those announcements of the specified category.

Return values:

id list	A list of announcement ids

`outbound_list_announcement` [`-category` *category*]

Description

The function gets a list of existing announcement that have a positive *outbound_total.* The *category* filters the list to only include those announcements of the specified category.

Return values:

id list	A list of announcement ids

`is_announcement_heard` *id*

Description

The function marks that this mailbox heard the announcement.

Return values:

`INVALIDID`	The id of the announcement is invalid.
`0`	The announcement has not been heard by the mailbox.
`1`	The announcement has been heard by the mailbox.

5.4 Published MMOs

Published MMOs are MMOs that are made public to the world. Anyone can publish a MMO, being logged in is not required. Once a MMO is published then other threads can get a handle on that MMO. Publishing a MMO can be selective on who it is published to.

Published MMOs are not persistent. When publishing an MMO a variable gets associated with the publication. When that variable is unset then the publication ends. Other threads must get the published MMO before the variable goes out of scope. But, once the thread has a handle on the MMO then they have a valid reference on that MMO to do as it pleases.

An example usage of Published MMOs is the following: A caller calls in and records a name and company recording. That recording gets published to the callee. The callee can get the recording to figure out if the callee wishes to talk to the caller, basicaly a form of call screening.

The following are the commands to manipulate announcements:

---

* `publish_mmo` *mmo_func common_key publish_var* [*mailbox*]...

Description

Publish the *mmo_func* to the specified *mailbox* list. If no *mailbox* list is specified then all this is published to everyone. If *mailbox* is `EVERYONE` then it explicitely states to publish the mmo to everyone regardless of other *mailbox* in the list. If *mailbox* is `LOGGED_IN` then the mmo is published only to those threads that are logged in. The *common_key* is used to create a unique key to represent the Published MMO. The value of *common_key* can be anything. The *publish_var* is the name of the variable that is created by this function to represent the lifespan of the publication. Once this variable goes out of scope then the MMO is no longer published. During the published time the, the variable represents another reference on the MMO.

Return values:

`BOXNOTEXIST`	*mailbox* is invalid
`MMONOTFORWARDABLE`	*mmo_func* is invalid

---

* `get_published_mmo` *unique_key mmo_var*

Description

Get the Published MMO as specified by the *unique_key* and create a new handle for the Published MMO that is named *mmo_var*.

Return values:

`KEYNOTFOUND`	*unique_key* is invalid

---

* `list_published_mmos`

Description

Get a list of Published MMOs that you have access to. What is returned is a list of *unique_key* that can be used by the get_published_mmo function to get the actual mmo. The returned list can be empty.

Return values:

unique_key list	A list of published mmo unique_keys

Chapter 6

Voice and Fax Devices

Voice devices and to a lesser extent fax and internet devices form the heart of Amanda Portal. Currently, the only voice boards supported by Amanda Portal are a certain set of Dialogic boards which support the SCBus and a certain set of Rhetorex boards. In addition, the Dialogic GammaLink SCBus fax resource boards are supported. These modules are Dialogic.DLL, RhetStd.dll, and Gamma.DLL. In addition, a SoundCard driver is available which provides essentially the same functionality as one telephone port plus a telephone. It is used for testing and demonstration purposes, and it is called Audio.DLL.

There are two sets of numbers assigned to devices in the system—ports and units—and they are not the same. Unit numbers are unique for each device within that device's class. That is, each VP device will have a different unit number but there may be an LS device with the same unit number. In addition to unit numbers, physical telephone ports are assigned unique *port numbers.*

When connecting devices on an SCBus, any device can broadcast to as many devices as possible but only listen to one device at a time. Figure 6.1 shows the concept. The VP device is listening to device LS_1 and devices LS_2 and LS_3 are listening to the VP device. It is possible for a device to listen to no one and send to no one too.

6.1 OOP Model

There are two approaches to use when defining Tcl commands: action-oriented and object-oriented. In the action-oriented approach, there is a separate command for each action taken by an object and the object is passed as the first parameter. That is, you would call `play $mmo` to play an MMO object. This works well when there is a large number of objects.

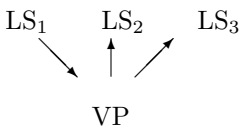

Figure 6.1: LS and VP device configuration

In the object-oriented approach, there is one command for each object and the name of the command is the name of the object. (In our implementation, a variable and the command name are bound together.) The first argument to the command specifies the command to execute. For example, you would say "`$vp play $mmo`" to play the MMO on the VP device.

The object-oriented approach lends itself well when the number of objects is small and the operations on the objects may vary. VP and MMO handles fit this description nicely. Also, Tcl allows you to add internal state to functions (through `ClientData`) but not variables. Therefore, using the object-oriented approach allows us to attach state to the function referenced by the `$vp` variable. We couldn't do this using the action-oriented approach. Currently we attach information about the VP or LS device in the state field. This state is actually a C++ object that we can operate on once inside C++.

6.2 Interconnection Limitations

Dialogic puts LS and VP devices on the same board or allows you to connect the devices on separate boards through the SCBus. This bus is a connector cable between two different boards that allows the boards to communicate with each other at high speed. Unfortunately, some Dialogic boards have limitations on the separate, simultaneous use of the LS and VP devices such that it is better to use a corresponding VP device with each LS device, rather than an arbitrary one. To accomodate this limitation, each LS and T1 device "knows" the unit number of its corresponding VP device, and has a `dsp_attach` command which checks out that particular VP device from the resource manager and establishes a full-duplex connection with it.

6.3 `Make_local` and Default Commands

Each object-oriented command has a command named `cmds` which lists all the commands that are valid for that device. If you type the command

```
$vp_handle cmds
```

you may get back something like

```
unit pause resume stop skip result play play_hold play_prompt speak
record pcpm beep load_prompt_set dialtone get_digits dial load_tones set_param
```

which lists all the commands that the `$vp_handle` command will take.

If you have a VP resource, you may not wish to keep typing

```
$vp_handle play mmo_funcs...
```

all the time. Instead it might be nice just to say

```
play mmo_funcs...
```

and use `$vp_handle` as the default VP device. The built-in `make_local` command creates a new command for each function that the handle passed as its argument supports. When the function is called, it internally uses the handle as its default argument. That is, `make_local` basically works as follows:

```
proc make_local {handle_var} {
    upvar 1 $handle_var handle

    foreach func [$handle cmds] {
        ## For every cmd this handle has, define a new command with
        ## an implicit $handle before it.
        proc $func {args} [list uplevel 1 [list $handle] [list $func] {$args}]

        ## Delete procedure upon variable exit.
        ## There should already be an unset trace on
        ## this variable for handle cleanup
        ## but we can add many more for each function we
        ## just added.
        uplevel [list trace variable $handle_var u [list _delete_cmd $func]]
    }
}

### Internal routine used by make_local.
proc _delete_cmd {func var index op} {
    catch {rename $func {}}
}
```

The `make_local` command will create a command like

```
proc cmd {args} {
    uplevel 1 {handle} {cmd} $args
}
```

Notice that `make_local` cleans up the procedures it creates when the variable leaves scope by creating an additional trace callback for each procedure that it creates. (This callback is a "lexical closure" for you Lisp fanatics.) The actual `make_local` command is a built-in command, not a proc.

`get_board_serial_number` *board*

Description

Returns the serial number of the indicated board. This function is available only when the Dialogic DLL has been loaded with the system and when the logged-in mailbox has the MONITOR privilege.

Return values:
Returns the indicated board's serial number.

Error codes:
`NOTINTEGER` *board*
`INVALIDVALUE` *board*

6.4 Network Device Commands

Each of the T1, LS, VP, and Fax commands can get internal errors generated by the Dialogic driver. If this is the case, a Tcl exception will be raised with an `errorCode` corresponding to the format

`BOARDERROR` *error_num*

and the Tcl result will be set to the board error message. Virtually every board related function can receive an error of this type, so these error codes are not listed.

To allocate an LS, VP, or Fax device, you'll need to use the `grab_res` routine discussed on page 38 or the `dsp_attach` command.

The network device (T1 and LS) commands are as follows. For simplicity, we use *ls* to represent any kind of network device, and note any differences that exist between the different device types which are supported in describing the command.

---

* *ls* `set_hook` *boolean*

Description

This function sets the phone on or off hook for the *ls* device. It is usually used when you are receiving a call or responding to a detected hangup.

Return values:
Empty string.

Error codes:
`NOTBOOLEAN` *boolean*

---

* *ls* `seize`

Description

This function seizes a line for an outgoing call. Call this before you make an outgoing call. On an analog line, this simply goes off hook and verifies that loop current is present. On a digital device such as a T1 line, the seize operation is verified digitally by the far end.

Return values:
Empty string.

---

* *ls* `disconnect`

Description

Actively hang up on any existing phone connection and insure that the call is terminated. It is the opposite of seize. For analog devices, the channel is held on hook long enough so that it doesn't look like a flash hook. This time is determined by the value of the `tmo_pickup` parameter in the Configuration Database. For T1 lines, of course, the release operation is verified digitally by the far end.

Return values:
Empty string.

---

* *ls* `connect` [`-half`] *resource_func*

Description

Connect the LS device to another resource. If `-half` is given, only a half duplex connection is set up; otherwise, a full duplex connection is set up. With a half duplex connection, the *resource_func* device listens to the *ls* device but not vice-versa. The use of `dsp_attach` is preferred over this command because it takes into account dependencies between the LS and VP devices.

Return values:
Empty string.

Error codes:
`CMDNOTEXIST` *resource_func*
`CMDNOTHANDLE` *resource_func*
`HWRONGTYPE` *resource_func*

---

* *ls* `dsp_attach` [`-timeout` *milliseconds*] *var*

Description

Each LS (and T1) device which lives on a board which has both telephone network interfaces and voice processing resources "knows" the unit number of its corresponding VP device. This command tells the network device to check out that specific VP device and then establish a full-duplex connection with it. The handle for the checked out VP device is put in *var*.

If `-timeout` is given, time out after the given number of milliseconds if the VP device isn't available.

Return values:
If `-timeout` *is given, then* `TIMEOUT` *is returned on a timeout. Otherwise, the return value is the empty string, and the var is set to the device's handle.*

Error codes:
`RESALLOC2BIG vp`
`NOTNONNEG` *milliseconds*

---

* *ls* `wait_off` *milliseconds*

Description

Wait for the other side to hang up. On an analog line, this function waits for loop current to drop. Some phone switches do not drop loop current when the far end hangs up, so take care in using this function in this situation—it will not return if reorder tone is detected. On a T1 line, it waits for digital indication that the far end has hung up.

If *milliseconds* is given, wait a maximum of *milliseconds* else wait forever.

Return values:

Empty string.	Success.
`TIMEOUT`	Waited more than *milliseconds*.

Error codes:
`NOTNONNEG` *milliseconds*

---

* *ls* `wait_on` *milliseconds*

Description

On an analog port, this function waits for loop current to start. On a digital line, it waits until the far end attempts to establish a connection.

If *milliseconds* is given, wait a maximum of *milliseconds* else wait forever.

Return values:

Empty string.	Success.
`TIMEOUT`	Waited more than *milliseconds*.

Error codes:
`NOTNONNEG` *milliseconds*

---

* *ls* `wait_ring` *milliseconds*

Description

This function is implemented by analog ports only. It causes the port to wait for ring voltage to be detected on the line. On a digital port, use the `wait_on` command to accomplish the equivalent function. The system must receive `n_rings` rings before the wait is satisfied, if this PBX parameter has been specified for this port.

If *milliseconds* isn't given, then wait forever.

Return values:

Empty string.	Success.
`TIMEOUT`	Waited more than *milliseconds*.

Error codes:

`NOTNONNEG` *milliseconds*
`OFFHOOK` *port*

---

* *ls* `unit`

Description

Return unit number of LS device. Devices of the same type have different unit numbers, but devices of different types may have the same unit number. This command won't return the `BOARDERROR` errorCode like the other commands will because it doesn't make calls to the device driver for the board.

Return values:

Unit number.

---

* *ls* `port`

Description

Return port number of LS device. Every network device in the system has a unique port number; *i.e.*, port numbers are not the same even for different device types. This command won't return the `BOARDERROR` errorCode like the other commands will because it doesn't make calls to the device driver for the board.

Return values:

Unit number.

---

* *ls* `set_screen_text` [`-box` *box*] [`-status` *text*]

Description

This function sets the text on the screen for the associated port. You can set either field or both at once. This routine won't return an `BOARDERROR` errorCode because it doesn't call the device driver functions.

Return values:
Empty string.

---

* *ls* `flashhook`

Description

Do a flashhook. In some environments, with some analog boards, this function actually does an Earth Recall function on the board (set during driver installation for some voice boards). This function is not supported on E1 boards. On T1 devices, the A and B bits are dropped for the duration of the `flashtm` parameter, or 500ms if that parameter is not specified in the Configuration Database.

Return values:
Empty string.

---

* *ls* `get_dnis`

Description

This function works only on GlobalCall network handles.

It returns the Dialed Number Identification String as returned by the GlobalCall driver for the associated port. If no DNIS information is available, then an empty string is returned.

---

* *ls* `get_ani`

Description

This function works only on GlobalCall network handles.

It returns the Automatic Number Identification string as returned by the GlobalCall driver for the associated port. If no ANI information is available, then an empty string is returned.

6.5 VP Commands

VP devices can perform long-term operations like playing a recording. Because of this, their operations may be interrupted by the detection of an exceptional condition, such as detection of certain incoming tones like a Fax CNG tone or TDD initiation tone, which indicate that special call processing is in order. We may also detect that the other party has hung up. Hangup detection is complex because there are so many ways that phone switches may indicate it to us:

1. On an analog port, loop current may be dropped. We look for this condition if the Configuration Parameter `hangup_supervision` is set to `true` for the port in the Configuration Database.
2. On digital ports, the port device will get a digital indication almost immediately if the other party hangs up.
3. The VP device, while playing or recording, may detect a tone or cadence which indicates that the caller has hung up, called a *reorder tone.* Or it may detect other special tones such as a Fax CNG tone.
4. The VP device may detect one or more DTMF digits which indicate hangup. Whenever digits are received, the VP device checks them against the setting of `dt_hangup` in the Configuration Database for the port to which the VP device has most recently been `dsp_attach`'d (remember that potentially the same VP device can be used with different network ports which are connected to different telephone switches which might send different hangup sequences).

In the first two cases, the network device will signal the VP device which was most recently `dsp_attach`'d to it, that it has detected hangup. In the latter two cases, the VP device itself does the detection. Either way, the VP device will then terminate whatever function it was performing and throw a HANGUP exception. If a Fax CNG tone is detected, a VP device will throw a `FAX_INITIATION` exception.

Here are the VP commands:

---

* *vp* `unit`

Description

Return unit number of VP device. Devices of the same type have different unit numbers, but devices of different types may have the same unit number. VP devices don't have port numbers. They don't stick out of the back of a machine.

Return values:
Unit number.

---

* *vp* `pause`

Description

Pause playing or recording of the current audio. You can resume playing or recording with the resume command later. This command should only be executed after you have started an asynchronous operation. You can also skip backward and forward while paused.

Return values:

Empty string.	Success.
NOOPINPROGRESS	No operation is currently in progress.

---

* *vp* `resume`

Description

Resume playing or recording. This command should only be executed after you have stopped an asynchronous operation.

Return values:
Empty string.

Error codes:
NOOPINPROGRESS

---

* *vp* `stop`

Description

Stop playing or recording on the specified device. This command differs from the pause command in that you cannot resume. This command should only be executed after you have started an asynchronous operation.

Return values:

Empty string.	Success.
NOOPINPROGRESS	No operation is currently in progress.

---

* *vp* `skip backward|forward`

Description

Skip forward or backward in the audio that is playing by "`skipby`" seconds. "`skipby`" defaults to five seconds or the value of the `PLAY_SKIP` box setting and can be set with the `-skipby` option on the play command line. This command should only be executed after you have started an asynchronous play operation. If you are currently paused, you will skip forward or backward and play will resume.

Return values:

Empty string.	Success.
NOOPINPROGRESS	No operation is currently in progress. This is not an exception because you want to a avoid a race condition. When you execute this command, play may have finished. This should not be an error.

Error codes:

errorCode	*Description*
NOMEANING	You tried to skip during a record operation.

---

* *vp* `result` [`-timeout` *milliseconds*]

Description

Returns the result of an asynchronous operation, such as a `play`, `record`, or `get_digits` operation. Normally, it waits forever for the result. If the `-timeout` command is given, only wait for the specified amount of time.

Return values:

Anything	Return result is operation dependent.
TIMEOUT	Waiting for result timed out.
STOPPED	A stop command was issued for the thread.

Error codes:

errorCode	*Description*
NOTNONNEG *milliseconds*	
HANGUP	
FAX_INITIATION	
FAX_ANSWER	
TONE *toneid*	The tone corresponding to *toneid* was heard. The mapping between toneids and tones is hardcoded in the system.

---

* *vp* `play` [`-async`] [`-clear`] [`-maxpause` *milliseconds*] [`-times` *n*] [`-delay` *milliseconds*] [`-pause` *pause_key*[*resume_key*]] [`-volume` *up_key down_key*] [`-speed` *fast_key slow_key*] [`-skip` *backward_key forward_key*] [`-skipby` *seconds*] [`-term` *key_string*] [`-fromend`] [`-noretain`] [*mmo_func*]...

Description

Plays a set of audio MMOs on the specified VP device. The MMO internal procedures names should be given (by dereferencing the MMO variables). The options are as follows:

`-async`	Play asynchronously. That is, return immediately.
`-clear`	Clear the DTMF buffer before playing. The DTMF buffer stores the digits the user typed in. This option is normally only used if you are playing back an error and want to discard what the user typed in.
`-maxpause` *milliseconds*	Maximum time that play may be paused. That is, if you pause the play and *milliseconds* expires, the play starts back up again automatically. The default is the setting `tmo_pause` in the Configuration Database. If this value isn't set, you can pause as long as you would like.

`-times`*n*	Repeats the playback for *n* times. The default is to play the MMOs once.
`-delay` *milliseconds*	When playback is repeated using `-times`, then playback will pause *milliseconds* between each repetition. The default is zero.
`-pause` *pause_key*[*resume_key*]	Pause if the user presses *pause_key*. Resume if the user presses *resume_key*. If *resume_key* isn't given, the *pause_key* is a toggle.
`-volume` *down_key up_key*	Adjust the audio volume if the user hits the specified keys. The space between the keys should not be given. It is simply here to separate the words *down_key* and *up_key*. Use a two character string.
`-speed` *slow_key fast_key*	Adjust the audio playback speed with the keys given. The two DTMF digits should actually be given back to back. For example: `-speed 70` would assign `7` as the slow-down key and `0` as the speed-up key.
`-skip` *backward_key forward_key*	Skip forward or backward by `skipby` seconds. `skipby` is set with the `-skipby` option. `skipby` defaults to 5 seconds or the value of the `PLAY_SKIP` box setting. As with `-volume` and `-speed`, the space between the keys is not actually specified in the command line; it is shown here only to separate the two words in the command description.
`-skipby` *seconds*	Set the number of seconds to skip forward or backward by.
`-term` *key_string*	Terminate playing when one of the keys in *key_string* is pressed. The default is all keys except for the skip, speed or volume keys. To not terminate on any keys, give the empty string.
`-fromend`	Start playing from the end. Playing is started `skipby` seconds from the end.
`-noretain`	Do not retain the key that terminated play in the DTMF buffer. If this option is not given, the key that terminated play is left in the DTMF buffer and will be processed by the next Tcl command that inspects the DTMF buffer.

Return values:

Empty string.	Play finished or `-async` was given and there was not an error.
key	A key was pressed.
`STOPPED`	A stop command was issued.

Error codes:

`NOTNONNEG` *seconds*
`CMDNOTEXIST` *func*
`CMDNOTHANDLE` *func*
`HWRONGTYPE` *func*
`HANGUP`
`FAX_INITIATION`
`FAX_ANSWER`
`TONE` *toneid*

---

* *vp* `play_hold` [`-async`] [`-clear`] [`-delay` *seconds*] [`-times` *number*] [`-term` *key_string*] [`-noretain`]

Description

Plays hold music. This command differs than the `play` command in that you can have the play loop over and over again and you can specify a delay between plays. The file(s) which are played are fixed; they are listed in the Configuration Database in the `hold_music` parameter.

The options listed mean the same thing as the `play` command with the exception of the following:

Return values:

Empty string.	Play finished or `-async` was given and there was no error.
key	*key* was pressed.
`STOPPED`	A stop command was issued.

Error codes:

`NOTNONNEG` *seconds*|*number*
`HANGUP`
`FAX_INITIATION`
`FAX_ANSWER`
`TONE` *toneid*

---

* *vp* `play_prompt` [`-async`] [`-clear`] [`-term` *key_string*] [`-noretain`] *prompt_set* [*prompt*]...

Description

Play the prompts. Prompts are prerecorded phrases in a file which is opened by the `load_prompt_set` command described below. The specific file to play the prompt from is specified by *prompt_set*. *prompt_set* is the returned value from `load_prompt_set`. Each prompt has a number. You give the sequence of the prompts to play on the command line. The meaning of the options is the same as the `play` command.

Return values:

Empty string.	Play finished or `-async` was given and there was no error.
key	*key* was pressed.
`STOPPED`	A stop command was issued.

Error codes:

`NOTNONNEG` *prompt*
`PROMPTNOTEXIST` *prompt*
`FAX_INITIATION`
`FAX_ANSWER`
`TONE` *toneid*
`HANGUP`
`LANGNOTSET`

---

* *vp* `record` [`-async`] [`-rate normal`|`high`|`wave16`] [`-mintime` *seconds*] [`-maxtime` *seconds*] [`-term` *key_string*] [`-clear`] [`-nobeep`] [`-append`] [`-hangup`] [`-noretain`] [`-notrim`] [`-silence` *seconds*] [`-initsilence` *seconds*] *mmo_func*

Description

Record audio to the indicated *mmo_func*. The options are the same as to the `play` command with the exception of

`-rate normal|high|wave16`	Record at a normal or high rate if `normal` or `high` is given. The actual rate is taken from the `rate_normal` and `rate_high` settings in the Configuration Database. MMOs in the `wave16` format are used by speech recognition, and they can be included in Internet e-mail messages with the `send_smtp_mail` command. If you are appending, this option is ignored.
`-mintime` *seconds*	If the recording is less than *seconds*, ignore the recording. The default is the parameter `minmsg` in the Configuration Database or 0 if that parameter is not set.
`-maxtime` *seconds*	If the recording is longer than *seconds*, then terminate the recording at *seconds* length and stop recording. The recording is saved at that length. Default: infinite.
`-nobeep`	Don't sound a beep to start recording.
`-append`	Append to the audio file rather than replacing it.
`-hangup`	Don't raise an exception on hangup. Simply return "`HANGUP`." This is useful for recordings which should be saved even if the caller hangs up.
`-silence` *seconds*	Terminate if *seconds* of silence is heard. This value overrides the parameter `tmo_silence` in the Configuration Database for silence termination. If `tmo_silence` is not set, the default is 5 seconds.
`-initsilence` *seconds*	Sets the initial silence to allow before termination. If given, then *seconds* seconds of initial silence is allowed. If sound is heard within this time, after `-silence` *seconds* recording will terminate. Useful for allowing a long lead time for someone to say something, but after they've said something, you wish to terminate quickly.
`-noretain`	Do not retain the digit that stopped recording in the DTMF buffer.
`-notrim`	Do not trim silence off the end of the recording when using `-silence`. Speech recognition does not work if you trim the silence off the end.
`-term` *key_string*	Terminate recording when one of the keys is pressed. The default is only the # key to help avoid talkoff.

Return values:

Empty string.	User gave `-async` and there are no digits in the DTMF buffer.
key	key was pressed.
`STOPPED`	A stop command was issued.
`HANGUP`	The user hung up. Only returned if `-hangup` is given.
`MINTIME`	The recording was ignored because it was too short.
`MAXTIME`	The maximum recording time was reached.
`MAXSILENCE`	The maximum amount of silence was heard.

Error codes:

`NOTNONNEG` *seconds*
`CMDNOTEXIST` *func*
`CMDNOTHANDLE` *func*
`HANGUP`
`HREADONLY` *func*
`FAX_INITIATION`
`FAX_ANSWER`
`TONE` *toneid*

* *vp* `pcpm` [`-rings` *number*] [`-async`] [`-after`]

Description

Perform "programmed call progress monitoring"; that is, listen for certain types of tones and return the result found. Example tones are busy signal, the fax answer tone, *etc.* `-async` works the same as the in the `play` command.

`-rings` says to return `NOANSWER` after *number* of rings. If `-rings` isn't given, then use the value of the `remote_rna` parameter in the Configuration Database. If this isn't set, the default is four.

The `-after` option specifies that when an answer is detected by hearing voice (this depends on the setting of the Configuration Parameter `use_pvc`), then wait until the voice stops before returning `ANSWER`. Otherwise, `ANSWER` is returned as soon as the voice is detected.

Return values:

Empty string.	You gave the `-async` parameter.
`BUSY`	A busy tone was detected. You usually hear this when the phone is off the hook.
`INTERCEPT`	Operator intercept was detected or call blocking.
`ANSWER`	Answer was detected.
`FAX_ANSWER`	A fax answer tone was heard.
`NOANSWER`	Phone rang but no answer.
`NORINGBACK`	No ringback was detected.
`TONE` *toneid*	

Error codes:

errorCode	*Description*
`NOTNONNEG` *rings*	
`HANGUP`	
`FAX_INITIATION`	This normally would not happen. Normally you call out and wait for the other side to answer using this call. When the other side answers, you hear a fax answer tone. You should never hear a fax initiation tone when the other side answers.

---

* *vp* `beep` [`-freq1` *frequency*] [`-freq2` *frequency*] [`-duration` *milliseconds*] [`-term` *string*] [`-dialtone`][`-busy`][`-ringback`][`-call_waiting`] [`-reorder`][`-override`][`-stutter_dialtone`] [`-recall_dialtone`][`-clear`][`-noretain`]

Description

Make a beep sound on the VP device. You can set the first and second frequencies with `-freq1` and `-freq2` You can set how long you want the beep to occur with the `-duration` option. The default is *freq1* of 500, *freq2* of 0, and *duration* of 100.

As with `play`, the `-term` flag indicates the list of digits which should terminate the function (the default is all digits), `noretain` indicates that terminating digits should be discarded from the digit buffer, and `-clear` indicates that any digits in the buffer should be cleared before beginning the function.

The options `-dialtone`, `-busy`, etc., cause a corresponding standard tone to be played. The use of these flags overrides the `-freq1` and `-freq2` flags.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *frequency|milliseconds*
`HANGUP`
`FAX_INITIATION`
`FAX_ANSWER`
`TONE` *toneid*

---

* *vp* `load_prompt_set` *prompt_set_name*

Description

Loads a prompt file into memory as specified by *prompt_set_name.* The directory location and extension should not be specified, just the name. This will return a value to be used by future calls to `play_prompt` to play a specific prompt within this prompt set. This function is similar to the now extinct `language` command. The idea is that now multiple prompt sets can be loaded at the same time for each language. Therefore, this function does not set the current language, nor does it source the prompt set's tcl file. It is now designed so that the prompt set's tcl file calls this function.

Still, a list of available languages is returned by the function `list_languages`, described on page 111.

Return values:
prompt_set

Error codes:
`LANGNOTEXIST` *prompt_set_name*

---

* *vp* `dialtone` [`-waittime`]

Description

Wait for one second of continuous non-silence, presumed to be a dialtone. If *waittime* is specified, then the function will wait that long to detect a dialtone; the minimum value of 2000 milliseconds is enforced silently. If no value is specified, then the function will wait the minimum time (two seconds).

Return values:

`1`	A dial tone was detected.
`0`	No dial tone was detected.

Error codes:
HANGUP
FAX_INITIATION
FAX_ANSWER
TONE *toneid*

---

* *vp* `set_digit_type` `mf` *or* `dtmf`

Description

This function changes the type of digits that the driver is to listen for. It does not flush any digits which may already be in the hardware or software digit buffer.

Return values:
Empty string.

---

* *vp* `get_digits` [`-clear`] [`-async`] [`-maxtime` *milliseconds*] [`-initsilence` *milliseconds*] [`-interdigitsilence` *milliseconds*] [`-maxdigits` *number*] [`-retain`] [`-keep`] [`-notrace`] [`-term` *key_string*]

Description

Listen for digits on a VP device. Some of the options are the same as in the `play` command. The ones that aren't are

`-maxtime` *milliseconds*	Listen for no more than *milliseconds* whether there is silence or not. This is the maximum total time for the operation. Default: 0 (unlimited time).
`-initsilence` *milliseconds*	*milliseconds* is the maximum initial silence before giving up. Default: 2000 (2 seconds).
`-interdigitsilence` *milliseconds*	If more than *milliseconds* occurs between a user typing a digit, assume the user is done typing digits. Default: 2000 (2 seconds).
`-maxdigits` *number*	Listen for a maximum number of digits. If you want to listen for a single digit, give 1.
`-retain`	Return the digits heard, but don't delete them from the DTMF buffer. Normally the digits are deleted from the DTMF buffer.
`-keep`	Keep the terminating digit. Normally, any digit that is in *key_string* is discarded and not returned (or put in the DTMF buffer).
`-notrace`	Suppress tracing the result of this function, for security reasons.
`-term` *key_string*	Terminate on the string given. The default is no key.

Return values:

Empty string.	`-async` was given or the operation timed out and no digits were heard.
Digits heard.	

Error codes:
FAX_ANSWER

errorCode
`HANGUP`
`FAX_INITIATION`
`TONE` *toneid*

---

* *vp* `dial` [`-mf` | `-pulse` | `-dtmf`] [*string*]...

Description

This function dials the string given. If multiple strings are given, they are concatenated together without intervening whitespace. The options tell how to dial. The default is `-dtmf`.

Return values:
Empty string.

Error codes:
`INVALIDVALUE` *string*

---

* *vp* `load_tones` *tone_set_filename*

Description

Load the tones from the specified *tone_set_filename* into the *vp* device. Some boards, such as Dialogic have consolidated tone sets where each VP device understands all the tones, and which are configured using the device driver. For these devices, this function is a no-op. Other boards, such as Rhetorex, may only understand a certain tone set for each VP device which needs to be configured at run-time. It is for the latter class of boards that this function is reserved.

Return values:
Empty string.

Error codes:
`FILENOTFOUND` *tone_set_filename*

---

* *vp* `get_mmo` *mmo_var*

Description

mmo_var is a writable mmo that will receive the last recorded sound that was recorded through the `recognize` function.

Return values:
Empty string on success. `NOTFOUND` *if there is no recording to retrieve* `RHETSTD_VOX_OPEN` *if it failed to open the mmo file* `RHETSTD_RECORD` *if no bytes were written to the mmo*

Error codes:
`MMONOTWRITABLE` *mmo_var*

---

* *vp* `init_fax`

Description

This function works only on a GlobalCall-based VP handles.

If the underlying board supports DSP-based faxing, and if a fax resource is available, then this function returns 1. Otherwise, it returns 0. Once this function has returned 1 for a given VP handle, then that VP device can execute the `send_fax` or `receive_fax` commands which are described on page 113.

6.6 Port Messages

A person who is talking on the phone using up a port is usually affected by the actions that they themselves do through talking and dtmf digits. However there is another action and that is from a client logged in the mailbox that is on the receiving end of the call. The port can advertize that it will listen for actions sent by the client and act on them. This concept is similar to the Call Queue, but the Call Queue handles a list of calls that are not yet talking to the callee. Port Messages are actions from the client to the port once the caller is talking to the callee. What is similar to Call Queue is the type of actions: `TRANSFER_BOX`, `HANGUP`, `HOLD`, and `VOICEMAIL`. The following commands are used to manipulate Port Messages.

---

* `enqueue_port_msg` *net_var mailbox var*

Description

This function will create a enrollment variable, *var*, that will allow the port to listen for commands from the client. The client must be logged into the same *mailbox*. And for security purposes the *net_var* of the port itself must be passed in. Once this function completes, use the `wait` command passing in the *var*, along with other variables to wait on, to listen for an action from the client. The following are results from the wait of this *var*:

`TRANSFER_BOX` *box [grt [grtBox]]*	Transfer the call to the indicated *box*. Optionally a greeting to play is specified. The m
`HOLD` *[grt [grtBox]]*	Place the call on hold. Optionally a greeting to play is specified. The mailbox of the gr
`VOICEMAIL` *box [grt [grtBox]]*	Send the call to voice mail for the indicated box. Optionally a greeting to play is specif
`HANGUP`	Disconnect the call.

Return values:
Empty string.

Error codes:
`MULTQATTACH` *mailbox*
`BOXNOTEXIST` *mailbox*

`list_enqueued_ports`

Description

This function is used to list the port numbers of the ports listening for commands from the current logged in mailbox.

Return values:
A list of port numbers.

`send_port_msg` *port* `hangup|transfer|voicemail|hold` [`-grt` *grt_num*] [`-box` *transfer_to*]

Description

This function is used to send a command message to the specified *port*. The possible commands are `hangup|transfer|voicemail|hold`. If *box* or `voicemail` then *box* must be specified to designate where to. The *grt* may be specified to play a greeting to the caller before the action is taken, usually to tell the caller what is going on.

Return values:
Empty string.

Error codes:
`MSGNOTFORW` *port*

6.7 Miscellaneous Commands

`get_port_status` [*port*]

Description

Returns the status of port *port*. If *port* is not specified, then information about all ports is returned, as an a-list of port numbers followed by the status a-list for that port.

Return values:
The return value is an a-list with the following keys:

`state`	`on-hook` or `off-hook`
`box`	box logged into port if someone is logged in.
`calls_taken`	Number of calls taken on port.
`route_info`	Route information for port.
`last_call_time`	Time last call was taken on port in GMT.

Error codes:
`NOTNONNEG` *port*
`PORTNOTEXIST` *port*

`get_ports_in_use`

Description

Returns a Tcl list of the ports currently off-hook.

`reset_port` [*port*]...

Description

Resets the ports given. If the port is not doing anything, such as playing some audio, this command is a no-op. You need the RESET_PORT privilege to execute this command, and it is available only when the Dialogic DLL has been loaded with the system.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *port*
`PORTNOTEXIST` *port*
`PERMDENIED`

---

* `list_languages`

Description

Returns a Tcl list of available languages to the system.

Return values:
language list.

---

* `name_to_digits` *name...*

Description

This function takes one or more names and returns a corresponding list of keypad encodings of those names, using the standard American keypad mapping of letters to digits.

Return values:
digits list.

Chapter 7

Fax Commands

Amanda Portal supports Dialogic's GammaLink fax resource boards via the `gamma` plug-in module. These boards are the CP/4-SC, CP-6/SC, and CP-12/SC, which provide 4, 6, and 12 fax resources, respectively. Boards can be combined to provide up to 60 fax resources in one system.

As with LS and VP devices, you must first acquire a fax resource from the resource manager via `grab_res`. Once you have this resource, you must connect it (full-duplex) to a telephone network resource, such as an LS or T1 device. You should have already established the telephone connection to the remote fax device, either by detecting a CNG tone (FAX_INITIATION) or by dialing out and using PCPM to detect a fax answer tone (FAX_ANSWER). Once this has been accomplished, you may release the VP device which was used to do the dial and PCPM operations and make the full-duplex connection between the fax resource and the network resource.

The fax resource devices, *fx*, provide the following member functions:

---

* *fx* `unit`

Description

Returns the unit number of the fax device.

Return values:
The unit number.

---

* *fx* `send_fax` [`-cover` *text*|*text_mmo_func*] [*text*|*mmo_func*]...

Description

Sends a fax on the given fax device. If `-cover` is given, use the given text for a cover page. You can send text, textual MMOs, or fax (TIFF/F format) MMOs from the fax device. When sending a fax, the "csid" (the phone number the fax is considered sent from) is retrieved from the fax driver. This number is printed on the recipient's page as required by law. You can override the fax driver value by setting the `fax_id` value in the Configuration Database.

Return values:
Empty string.

Error codes:
`CMDNOTEXIST` *func*
`CMDNOTHANDLE` *func*
`HWRONGTYPE` *func*
`HANGUP`
`FAX_INITIATION`
`FAX_ANSWER`
`TONE` *toneid*

---

* *fx* `receive_fax` *mmo_func*

Description

Receives a fax on the specified fax device. *mmo_func* should be an MMO previously allocated with `create_mmo`.

Return values:

`NOFAXRECEIVED`	Receiving a fax failed. Perhaps the other side hung up before sending the fax or an error occurred during the transmission.
csid	The calling station id (csid) is returned on success. Normally this is the phone number of the station sending the fax.

Error codes:
`CMDNOTEXIST` *func*
`CMDNOTHANDLE` *func*
`HWRONGTYPE` *func*
`HREADONLY` *func*
`FAX_INITIATION`
`FAX_ANSWER`

In addition, see the `queue_fax` command described on page 200.

Chapter 8

Internet E-Mail

Amanda Portal implements POP3 client and server code and E-SMTP client and server code. The client code in each case is accessible at the Tcl level, while the server code runs automatically. All four features are enabled by loading the POP DLL. These features can be used as follows:

POP3 client This facility is useful for implementing a way to play back waiting e-mail, which remains stored on the POP3 server, to a user. Text to speech can be used to read mail which is in ASCII, while of course voice enclosures can be played directly.

SMTP client This feature allows you to create applications in which users originate Internet e-mail messages which contain voice components which they record via the telephone. It can also be used in a Notify template to transfer all of a user's messages to an SMTP/POP server so that the user receives all forms of messages in one place. The Notify template could control whether the messages were then deleted from Amanda Portal or were kept in both places.

POP3 server Another method of accessing Amanda Portal messages via a standard e-mail client is to tell the client to access Amanda Portal as a POP3 server. Amanda Portal will automatically translate all messages stored in the user's mailbox into standard MIME-format messages, so the POP client is unaware that they were originally voice and fax mail messages.

SMTP server The purpose of the SMTP server is mainly for when a user sends Internet mail to an outside recipient. If that recipient then tries to reply to the message, the SMTP server code exists so that Amanda Portal can receive the reply and store it in the user's mailbox. Currently, we recognize MIME and VPIM version 1.0 messages containing voice (in various formats) and fax (TIFF-F) enclosures as well as plain text.

POP servers assign message numbers to each message. You can delete messages by these message numbers. When you delete a message, the message isn't actually deleted on the POP server until the POP `quit` command is executed. Once a message is deleted, you cannot refer to it anymore and commands that operate on that message number won't work.

Here are the commands related to sending and receiving Internet e-mail:

`send_smtp_mail` [`-host` *host*] [`-subject` *subject_MMO*] [`-format` *msg_format*] [`-mmos` *MMO_list*] [`-from` *from*] [`-urgent`] [`-private`] *recipient*...

Description

Send e-mail to the indicated *recipients* If `-host` isn't specified, then the mail is posted through the host specified by the `smtp_host` parameter in the Configuration Database. You can give a subject line with `-subject`, whose MMO argument must contain plain ASCII text to conform to RFC-822. The default is not to set a subject. You can specify the contents of your message with zero or more MMOs. The *msg_format* argument specifies the category of MIME message that you want to create and the format of the audio attachments, if any. The available choices are `wave16` and `vpim`. If `-format` is not given, then a standard MIME message containing audiobasic sound files will be generated.

The `From:` field of the generated message will normally default to *mailbox*`@`*hostname*. The *mailbox* portion can be overriden by the `smtp_from` configuration setting, and similarly the *hostname* can be overriden by the `smtp_hostname` configuration parameter. However, all of this is superceded if the `-from` argument is used.

Return values:
Empty string.
`HOSTNOTEXIST` *host*
`CONNREFUSED` *host*
`USERNOTEXIST` *recipient*

Error codes:
`CMDNOTEXIST` *func*
`CMDNOTHANDLE` *func*
`HWRONGTYPE` *func*
`PROTOCOLERROR`

---

* `connect_to_pop_server` *user password host var*

Description

Connect to POP server *host* as *user* with *password* and return a handle in *var*. You can then use *var*'s member functions to get and delete messages.

Return values:
Empty string.
`HOSTNOTEXIST` *host*
`CONNREFUSED` *host*
`LOGINFAILED` *user password*

Once the variable is set by `connect_to_pop_server`, the handle's member functions are as follows:

---

* *hfunc* `stat`

Description

Returns two quantities: the number of undeleted messages on the server and the total number of bytes they represent.

Return values:
An a-list is returned. It will contain two keys. The first key `undeleted_count` *contains the number of undeleted messages on the server. The second value* `undeleted_bytes` *contains the number of bytes consumed by the undeleted messages.*

Error codes:
`CONNBROKEN`

---

* *hfunc* `list` [*msg_num*]

Description

Returns a list of the message numbers and their size in bytes. If *msg_num* is given, only the message number and size for the message given are returned.

Return values:
An ordered a-list is returned. The first part of each "2-tuple" is the message number and the second part is the size of that message in bytes.

Error codes:
`NOTNONNEG` *msg_num*
`MSGNOTEXIST` *msg_num*
`CONNBROKEN`

---

* *hfunc* `uniq_id_list` [*msg_num*]

Description

Returns a unique id list for all the non-deleted messages or a unique id list for the message given by *msg_num*. These unique ids are persistent and should be the same even across two different connections. This is not true for message numbers.

Return values:
An ordered a-list is returned. first part of each "2-tuple" is the message number. The second part if the unique id for the message.

Error codes:
`NOTNONNEG` *msg_num*
`MSGNOTEXIST` *msg_num*
`CONNBROKEN`

---

* *hfunc* `get_msg` *msg_num*

Description

Returns the message given by *msg_num.*

Return values:
The text of the message is returned.

Error codes:
`NOTNONNEG` *msg_num*
`MSGNOTEXIST` *msg_num*
`CONNBROKEN`

---

* *hfunc* `get_msg_top` *msg_num n*

Description

Returns the top n lines of a message. All the message headers are returned followed by n body lines.

Return values:
The headers and first n body lines are returned.

Error codes:
`NOTNONNEG` *msg_num*|*n*
`MSGNOTEXIST` *msg_num*
`CONNBROKEN`

---

* *hfunc* `delete_msg` *msg_num*

Description

Delete the message indicated by *msg_num*. As mentioned earlier, the message is not actually deleted by the pop server until a `quit` command has been issued; the `undelete_msgs` member command may be used to revive deleted messages until then.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *msg_num*

errorCode
`MSGNOTEXIST` *msg_num*
`CONNBROKEN`

---

* *hfunc* `undelete_msgs`

Description

Mark deleted messages as undeleted on the POP server.

Return values:
Empty string.

Error codes:
`CONNBROKEN`

---

* *hfunc* `quit`

Description

Remove all messages marked for deletion and close TCP connection. Simply `unset`ing the *hfunc* variable will drop the connection to the server without "expunging" any deleted messages there, in accordance with the latest POP3 RFC. Therefore, clients should issue the `quit` command first if they want to insure that the deleted messages are actually deleted.

Return values:
Empty string.

Error codes:
`CONNBROKEN`

There is one more function that becomes available when the `pop.dll` module is loaded:

---

* `parse_mail_msg` *text msg_var*

Description

Parse the text given as an e-mail message and fill in *msg_var*. *msg_var* has the same structure as a message returned by `get_next_msg` (see page 72), with the addition of one field: `preferred_from`. This field is set to the human name found in the `From` field of the message, or to an empty string if no display name is present there.

If the mail message contains audio MIME parts, they will be extracted as audio MMOs. If the message contains textual MIME parts, they will be extracted as textual MMOs.

Return values:

Empty string.	Success.
`PARSEERROR`	Couldn't parse the text.

Chapter 9

Serial Devices

The serial ports on a computer are a limited resource that is controlled by the resource manager much like the limited VP and LS devices. Amanda Portal can use serial ports, for instance, to integrate with certain types of phone switches (see chapter 14).

Like VP and LS devices, the resource manager creates a "handle" with member functions for serial devices. To make a serial port available as a resource, you must create a global Configuration Database parameter "com*n*" with a value equal to a string of settings similar to the DOS "`mode`" command. That is

```
9600,n,8,1
```

or

```
baud=9600 parity=N data=8 stop=1
```

will both work. When the system starts up, it opens any such com*n* devices and initializes them according to these settings. Normally, you will not have to change the baud rate, parity, etc., settings at runtime, though there are member functions listed below which can do so.

Here are the member functions associated with serial ports:

---

hfunc `baud` [*rate*]

Description

Sets the rate to *rate* or returns the current rate.

Error codes:
`NOTNONNEG` *rate*

errorCode
`INVALIDVALUE` *rate*

---

hfunc `parity` [*value*]

Description

Sets the parity to *value,* which must be one of `odd`, `even`, `mark`, `space`, or `none`, or returns the current setting.

Error codes:
`INVALIDVALUE` *value*

---

hfunc `binary` [*boolean*]

Description

Sets whether the port ignores null characters or returns the current setting. When reading strings with the `read` function, this parameter should be set to "`true`." The `gets` command eliminates any null bytes regardless of the setting of this parameter. Default: `false` (ignore null characters).

Error codes:
`NOTBOOLEAN` *boolean*

---

hfunc `flow_out` [*value*]

Description

Sets the output flow control to *value,* which must be one of `none`, `xon`, `cts`, or `dsr`, or returns the current setting.

Error codes:
`INVALIDVALUE` *value*

---

hfunc `flow_in` [*value*]

Description

Sets the input flow control to *value,* which must be one of `none`, `xon`, `rts`, `rts_handshake`, or `rts_toggle`, or returns the current setting.

Error codes:
`INVALIDVALUE` *value*

---

hfunc `stop_bits` [*bits*]

Description

Sets the number of stop bits to *bits,* which must be one of `1`, `1.5`, or `2`, or returns the current setting.

Error codes:
`INVALIDVALUE` *bits*

---

hfunc `data_bits` [*bits*]

Description

Sets the number of data bits to *bits,* which must be from `4` to `8`, or returns the current setting.

Error codes:
`NOTNONNEG` *bits*
`INVALIDVALUE` *bits*

---

hfunc `gets` [`-timeout` *milliseconds*]

Description

Read newline terminated text from the port. The terminating carriage-return/line-feed combination is not returned and null characters are ignored. If `-timeout` is given, time out after that many milliseconds; otherwise, wait forever.

Return values:

`TIMEOUT`	Timed out.
value	Value retrieved.

Error codes:
`NOTNUMBER` *milliseconds*

---

hfunc `getc` [`-timeout` *milliseconds*]

Description

Return a single character from the input stream. Default timeout is infinite.

Return values:

`TIMEOUT`	
Empty string.	Null character read.
character	Non-null character read.

Error codes:
`NOTNONNEG` *milliseconds*

---

hfunc `write` *string*

Description

Write the string to the serial port.

Return values:
Empty string.

---

hfunc `read` *max*

Description

Read a string of `max` characters from the serial port.

Return values:
The string is returned in ASCII hex. That is, a string of length $max * 2$ *is returned. The return value is in hex so you can decipher null characters.*

Error codes:
`NOTNONNEG` *max*

Chapter 10

Miscellaneous Databases

10.1 List Mapping Database

The list mapping database is a general persistent store for mappings from keys of the form "(*box*,*key*)" to a list of string values. The meaning of the *key* values is left up to the Tcl code. The main purpose of this database is to support mailing lists in Amanda Portal, where *key* would be something like "`List 3`" and the string values would be a list of box numbers. There are no privileges needed to use this store except for the box ancestor relationship restrictions that permeate Amanda Portal.

There is a special case in the list mapping database though. If *key* is `copy_to`, then you must have the COPY_TO privilege to change the mapping. This list is used internally to determine whether messages for boxes should be copied to other boxes on receipt. In addition to the COPY_TO privilege, the regular ancestor relationship restrictions in Amanda Portal apply, so you cannot modify the `copy_to` of a non-descendant even if you have the privilege.

Here is a list of the list mapping functions:

`add_lmapping` [`-box` *box*] *key* [*value*]...

Description

Add a set of values to a list mapping (lmapping). This adds new values to the list corresponding to the index "(*box*,*key.*)" It does not delete any previous values already in the list. Any duplicate values are silently ignored.

Return values:
Empty string.

Error codes:

errorCode	*Description*
`NOTNONNEG` *box*	

errorCode	*Description*
BOXNOTEXIST *box*	
PERMDENIED *box*	
NOTNONNEG *value*	Only received if using the copy_to key. Values for this key must be box numbers.
INVALIDVALUE *value*	

---

delete_lmapping [-box *box*] *key* [*value*]...

Description

Delete a set of values from a list mapping. This only deletes the values given and leaves the other values in the list. If you want to delete all the values in the list (essentially deleting the mapping), see the set_lmapping call.

Return values:
Empty string.

Error codes:
NOTNONNEG *box*
BOXNOTEXIST *box*
PERMDENIED *box*
INVALIDVALUE *value*
VALUENOTEXIST *value*

---

* get_lmapping [-box *box*] *key*

Description

This gets the value of the list mapping at the specified index.

Return values:
A Tcl list is returned with the list value. If there is no such index, an empty Tcl list is returned. Therefore, there is no difference between the list mapping not existing and being empty. They are one and the same.

Error codes:
NOTNONNEG *box*
BOXNOTEXIST *box*
INVALIDVALUE *value*
BREQNLOGIN

---

* get_lmapping_attendance [-box *owner_mailbox*] *member*

Description

This gets all the lists that this *member* belongs to. If *owner_mailbox* is specified then just the lists of that mailbox is searched.

Return values:
A Tcl alist is returned. The first part is the owner_mailbox. The second part is a Tcl list of list keys that the member is within. An empty result is return if member does not belong to any list.

Error codes:
`INVALIDVALUE` *value*

`set_lmapping` [`-box` *box*] *key* [*value*]...

Description

This sets the value of the list mapping to the indicated values. Old previous values *are* deleted. This function also doubles as the list mapping deletion function, since setting the list to empty and deleting the list are equivalent.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`INVALIDVALUE` *value*

---

* `get_lmapping_keys` [`-box` *box*]

Description

Get all the keys for the list mapping for the specified box or the current box if logged in.

Return values:
The keys for all the lmappings as a Tcl list.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`INVALIDVALUE` *value*
`BREQNLOGIN`

Another class of commands deals with persistent values. These values may be set, enumerated, or increemented/decremented atomically. The name of each value is a free-form string, as is the value, though of course if the value is to be incremented, then it must be an integer.

---

* `get_global` [`name...`]

Description

This function returns an a-list of the global variables, or keys, and their values. If no `name`s are specified, then all defined global variables are returned, and otherwise, only the requested ones are returned.

Error codes:

errorCode	*Description*
`INVALIDKEY`	The specified key is not currently defined.

---

* `increment_global` [`-by` *value*] `name...`

Description

This function increments the value of one or more of the permanent global keys. If more than one key is specified, then all are incremented together atomically. If a key is specified which does not exist, then it is treated as if it did exist with the value 0.

The option `-by` can be used to specify the integer *value* by which the variables are to be incremented. It defaults to 1.

Error codes:

errorCode	*Description*
`INVALIDKEY`	The specified key is not currently defined.
NOTINTEGER	The value of the specified variables is not an integer.

---

* `set_global name value...`

Description

This function can be used to set the value of one or more permanent global keys. If more than one key is specified, then all the keys are set together atomically. Any previous value associated with any specified key is lost.

`delete_global name...`

Description

This function can be used to delete one or more keys from the permanent global variables list.

This function can be called only by interpreters executing with the GLOBAL_MONITOR privilege.

Error codes:

errorCode	*Description*
`INVALIDKEY`	The specified key is not currently defined.

10.2 Trie Mapping Database

When you call into a phone system, you may not know a person's extension. Instead, you may only know a person's name. Therefore, we need a way of mapping from a person's name to the set of box numbers with that name. The only input device the user has at this point is his phone, so we need a way of mapping from a set of digits entered by the user to a set of box numbers (hopefully one).

When you first call in, you will hear a menu entry that tells you something like, "If you don't know the person's extension, type in the first three digits of their last name." When you do this, the system looks up the digits typed on the keypad in the "411 Directory" and maps those digits to the box numbers of the people with that name. Usually only one name will match and you will get that person's box (which is then used to find the person's extension).

"Tries" are data structures in computer science. When given a set of strings, "tries" find the shortest string match that uniquely identifies a string.

Tries solve the mapping problem from a fragment of the user's name to his extension. The user is prompted to type in the letters of the person's last name and the trie tree is traversed looking for matches. If a single match is found, then the value is looked up and returned. For example, if the value is the user's extension, then this value is spoken back to the user. If multiple matches are found, all the names found are read back to the user.

Since the user's only input is often a phone, two entries are actually installed in a trie tree whenever a new entry is installed: one entry is installed for the alphanumeric text itself and another entry is installed for the numbers on the keypad that correspond to the letters in the input key. When you query for the values associated with a key, you can query based on the alphanumeric or telephone numeric value. The "separation" between the alphanumeric and numeric entries is known by the system though, as we will see later.

There can be many trie databases, each separate from one another. To modify any trie database, you need the CHANGE_TMAPPING privilege. To create and destroy any trie database, you need the CREATE_TMAPPING privilege. One trie database is special though: the "`411`" trie database. This database has special semantics known to the system. With this database, the values must be box numbers, the boxes must exist and you can only change values for your box number or descendent box numbers. Also, when a box is deleted, any values are deleted from the `411` trie database automatically.

Here are the trie functions:

`create_tmapping` *map_name*

Description

Creates a new trie mapping with the name *map_name*. Initially the map has no keys. You must have the CREATE_TMAPPING privilege to execute this call.

Return values:
Empty string.

Error codes:
`MAPEXISTS` *map_name*
`PERMDENIED`

`destroy_tmapping` *map_name*

Description

Destroy a trie mapping database. All the entries and the map itself are destroyed. You must have the CREATE_TMAPPING privilege to execute this call. Only the superbox can destroy the `411` trie mapping database.

Return values:
Empty string.

Error codes:
`MAPNOTEXIST` *map_name*
`PERMDENIED`

`add_tmapping` *map_name* [*value*] *key*...

Description

Add a new *value* and the list of *key*s to the database if the key doesn't exist in the database already; otherwise, append the *value*s to the list associated with the *key* already in the database. *key* must be alphanumeric. If `key` contains letters, then two "*key*s" are actually entered into the database: one with the alphanumeric name and one with only the numeric name corresponding to the alphanumeric name on the telephone keypad. These two keys are known as "internal keys." If *key* contains digits only, then only one "internal key" is entered into the database.

If the *map_name* is "`411`," then the *value*s must be box numbers and the box numbers must correspond to either your box or a descendent box. These entries will automatically be reclaimed on box deletion.

You must have the CHANGE_TMAPPING privilege to execute this call.

Return values:
Empty string.

Error codes:

errorCode	*Description*
`INVALIDKEY` *key*	*Key* isn't alphanumeric. 411 map only.
`INVALIDVALUE` *value*	Received when you try to insert a non-numeric value into the 411 map.
`PERMDENIED`	
`MAPNOTEXIST` *map_name*	
`BOXNOTEXIST` *box*	`411` trie mapping database only.

`delete_tmapping` *map_name value key...*

Description

Delete the entry containing the associated *key* and *value*. If *key* maps into two "internal keys," then the value is removed from both lists corresponding to the "internal keys." When the internal list corresponding to an "internal key" becomes empty, the "internal key" itself is deleted. You need the CHANGE_TMAPPING privilege for this call.

Return values:
Empty string.

Error codes:

errorCode	*Description*
`PERMDENIED`	
`INVALIDKEY` *key*	Key isn't alphanumeric.
`KEYNOTEXIST`	Key does not exist.
`VALUENOTEXIST`	Value does not exist on the key given.
`MAPNOTEXIST` *map_name*	
`INVALIDVALUE` *value*	

`delete_tmapping_by_value` *map_name value*

Description

This function searches the lists of all the "internal key's"s values and removes any entries containing *value*. If the internal list becomes empty, the "internal key" is deleted also. This function is useful for deleting all keys associated with a box, if the *value* is a box. You need the CHANGE_TMAPPING privilege for this call.

Return values:
Empty string.

Error codes:
`PERMDENIED`
`MAPNOTEXIST` *map_name*
`INVALIDVALUE` *value*

---

* `query_tmapping` [`-alphanumeric`] *map_name key_fragment*

Description

Query the trie mapping database and try to find a match. The alphanumeric and numeric portions of the tree are known to the system. Normally this function only searches the numeric portion of the tree. If you want search the alphanumeric portion of the tree, use the `-alphanumeric` option.

Return values:
An a-list is returned where each index is a key that matched and the value is a list of values for that key. If the a-list contains more than one entry, then the key_fragment was ambiguous and it matched multiple keys. If the list for an index contains more than one entry, then there are multiple values for that key. (For example, there may be multiple boxes with the name "Bob.") If the a-list returned contains no entries, then nothing matched.

Error codes:
`INVALIDKEY` *key_fragment*
`MAPNOTEXIST` *map_name*
`INVALIDVALUE` *value*

---

* `get_tmapping_keys` *map_name value*

Description

Gets all the keys containing *value* somewhere in the key's list. Both the alphanumeric and numeric keys are returned.

Return values:
The keys are returned as a Tcl list.

Error codes:
`MAPNOTEXIST` *map_name*
`INVALIDVALUE` *value*

---

* `list_tmapping_databases`

Description

This functions returns a list of all the trie mapping databases in the system.

Return values:
A Tcl list of all the trie mapping databases in the system.

Chapter 11

The Configuration Database

The Configuration Database persistently stores configuration information in four distinct categories:

global Parameters that are global to the system.

port Parameters that are specific to a port on a board.

extension Parameters that are specific to an extension in the system.

pbx Parameters that are specific to a certain type of PBX in the system.

The last three types of parameters require some type of *id* to identify the port, extension or PBX in question. Global parameters don't require an *id*. Non-global parameters come in two forms: a "short form" which is a 1-tuple (*e.g.*, `n_rings`) and a "long form" which is a 2-tuple (*e.g.*, "`n_rings 2`" or "`n_rings *`"). The long form has a special "wildcard" form specified with a '*' which means all ports, extensions or PBXes (the "`n_rings *`" form in the previous sentence). Non-wildcard forms are considered more specific than wildcard forms and take precedence over them.

The parameters in these categories are pretty much fixed when the system is running. We'll see some other parameters in the next section that vary as the system is runs (such as the message waiting light on/off state). The parameters in this section require the CONFIGURATION privilege to change while the parameters in the next section (such as the message waiting light) do not require you to have the CONFIGURATION privilege.

Each of the categories of parameters are separate except for the port and pbx databases. If a key does not exist in the port database, the pbx database is checked for the parameter value. To determine which pbx settings to check for, the special setting `pbx` is checked in the port database to determine the id of the pbx database entry to check. For example, suppose we have the following entries:

PBX database	
`tmo_dtwait Panasonic`	`300`
`tmo_dtwait Partner_II`	`500`

Port database	
`tmo_dtwait 3`	`400`
`pbx 4`	`Panasonic`
`pbx *`	`Partner_II`

Port 3 will have a `tmo_dtwait` of 400 milliseconds. Port 4 will have a `tmo_dtwait` of 300 milliseconds (because it is on a Panasonic switch) and all other ports will have a `tmo_dtwait` of 500 milliseconds and are on a Partner II switch.

Here are the Configuration Database functions:

---

* `get_param` [`-id` *id*] `global`|`port`|`extension`|`pbx` [*key*]...

Description

This function gets the values of all the parameters with the keys given. *Key*s should be given in short form. The `-id` option may not be given if the type is `global` and must be given if the type is not `global`. This is the main function for looking up a single value. When looking up port values, the pbx database is searched as described above.

Return values:
An a-list containing the values is returned. The keys are always returned in short form and you cannot tell by the return value whether a wildcard or non-wildcard key matched. That is, if you give the command

```
get_param -id "Panasonic KX-T308/616" pbx foo_param
```

you may get back the a-list "`foo_param bar`*" and you won't know whether there is an "*`foo_param "Panasonic KX-T308/616"`*" or "*`foo_param *`*" key in the database. Also notice that the id need not be a number. For PBXes, the id is a string. If the key does not exist, the entry won't be set in the returned a-list.*

---

* `get_param_by_long` `global`|`port`|`extension`|`pbx` [*key*]...

Description

This function takes and returns parameters using the long form rather than the short form as in the previous function. That is, you can pass in "`n_rings 19`" and "`n_rings *`" and get back two values if there are two different values for these keys in the database. If you give no keys, you will get back nothing. To get all the keys, nest a call to `get_param_keys` like

```
array set foo [eval get_param_by_long port [get_param_keys port]]
```

The pbx database *is not* searched if the `port` category is used. This function is useful for getting all the values in a certain category.

Return values:
An a-list is returned with the keys in long form. This allows you to iterate over all the keys in the database.

Error codes:
`KEYNOTEXIST` *key*

---

* `get_param_keys global|port|extension|pbx|help`

Description

This function returns all the long key names for all the parameters of a certain type. The pbx database *is not* searched if the `port` category is used.

Return values:
A regular Tcl list with the keys in long form are returned.

`set_param -table global|-table port|-table extension|-table pbx` [*key value*]...

Description

This function allows you to set parameters. Parameters keys must be specified in long form for non-`global` parameter keys. You need the CONFIGURATION privilege to execute this call. You can add new unknown parameters to the system but the interpretation of these new parameters is left up to your Tcl code.

Return values:
Empty string.

Error codes:
`PERMDENIED`

`delete_param global|port|extension|pbx` [*key*]...

Description

This function deletes a set of parameters from the system. All *key*s should be given in long form. You must have the CONFIGURATION privilege to execute this call.

Return values:
Empty string.

Error codes:
`KEYNOTEXIST` *key*
`PERMDENIED`

Each parameter name has help text associated with it. The help text describes what the key means and is displayed on the screen to the user when he is looking at a certain key in the Configuration Database. The help text is not divided up into four individual databases like the regular Configuration Database. If you have two keys with the same name in the `global` and `port` databases for example, they must have the same help text. The help text manipulation functions are as follows:

---

* `get_param_help_text` [*key*]...

Description

Gets the help text for the specified *key*s.

Return values:
An a-list is returned with the key and help text associated with the key. If a key doesn't exist, that key value pair is not returned.

---

* `get_param_help_keys`

Description

Returns all the help key names.

Return values:
A Tcl list of the key names is returned.

`set_param_help_text` [*key value*]...

Description

Sets the help text for the specified keys. You need the CONFIGURATION privilege to use this function.

Return values:
Empty string.

Error codes:
`PERMDENIED`

For a list of all the parameters recognized by the system, see "The Amanda Portal Administration Guide."

Chapter 12

Triggers

There are three kinds of triggers in Amanda Portal: those that fire when a certain time is reached, those that fire when a certain condition is met, and those that fire when some data changes. These are known as autoschedules, notify records, and dtriggers (data triggers). Autoschedules are basically used for time related events like causing any phone calls to go to your voice mail after 5PM. Notify records are used, for example, to schedule jobs on your behalf when you receive a message. Dtriggers are used for data change events like notifying an Amanda client when some data in Amanda Portal changes.

12.1 Autoschedules

Autoschedule records are stored in a special database known to the system. They basically list, on a per-mailbox basis, what action is to be taken (Tcl code to be executed) and when that code should be executed. The code to execute may simply be the name of a Tcl proc, and often this procedure is defined via the "`unknown`" mechanism—that is, it is automatically loaded into the Tcl interpreter when you first attempt to use it and it's not already defined. When the code is executed, it runs in its own interpreter logged into the box to which the autoschedule record belongs.

The autoschedule database contains, on a per-mailbox basis, one record for each automatically scheduled command. Each record is assigned a schedule number by the user. Schedule records are created and updated by the `schedule_set` command. All the schedule records for the current box, or for a descendant box, can be obtained via the `schedule_list` command, and they can be deleted via the `schedule_delete` command. The complete settings for a particular autoschedule can be obtained using the `schedule_get` command.

`schedule_set [-`*box*`]` *sched_num field value...*

Description

This command can be used to create or modify autoschedule records for the current mailbox or for another box by specifying `-`*box*. The *field*/*value* pairs consist of one or more of the following fields in the database:

`INITTIME` This field specifies the first time that the autoschedule record should be executed. The time is specified as a coordinated universal time value. Normally, the `INITTIME` will specify a time in the future, but that is not strictly necessary.

`EXEYEAR` After the initial execution, autoschedules may repeat on a regular basis. The `EXEYEAR`, `EXEMONTH`, etc., fields specify that it is to repeat at an interval of every so many minutes, days, hours, months, and/or years. These values can be combined so that a schedule is repeated every one month and one day, for example. The default for each of these values is zero, so that if only an initial time is specified, then the schedule will run once at that time.

`EXEMONTH` See `EXEYEAR`.

`EXEDAY` See `EXEYEAR`.

`EXEHOUR` See `EXEYEAR`.

`EXEMINUTE` See `EXEYEAR`.

`METHODON` When an autoschedule record's time to execute comes, the `METHODON` value specifies what Tcl command is supposed to be executed. Often this command will come from the persistent store of pre-defined commands. But it will only execute if has the highest `PRIORITY` at this time. The value specified for this parameter cannot be an empty string. When creating a new schedule, this value is required.

`DURYEAR` Once the `METHODON` command has been executed, and if the `DURYEAR`, `DURMONTH`, `DURDAY`, `DURHOUR`, and/or `DURMINUTE` fields are non-zero, then the schedule is considered to be still in force. This feature is used in combination with the `METHODOFF` field to specify an action which is to occur some time after the `METHODON` action, usually to undo the `METHODON` action. The duration time must be earlier than the time of the next execution of the method, unless this schedule is only to execute once.

`DURMONTH` See `DURYEAR`.

`DURDAY` See `DURYEAR`.

`DURHOUR` See `DURYEAR`.

`DURMINUTE` See `DURYEAR`.

`METHODOFF` See `DURYEAR`.

`PRIORITY` If an autoschedule record is considered to be executing, then it will suppress the execution of any other autoschedule records of lower priority during that time period. The priority value is an arbitrary non-negative integer. This feature is usually used to create "holiday" schedules which will override any normal autoschedules which exist for the mailbox.

`PROTECTION` Allows a parent to create a schedule record for a child and to prevent that child from being able to change or delete the record, even if the child normally has the privilege to change its autoschedule records.

`SUN...SAT` Autoschedule records may be allowed to run only on certain days of the week. The `SUN` through `SAT` fields can be set to boolean values (`true` or `false`) to indicate whether the schedule is allowed to run on those days. When a schedule "attempts" to execute and finds that it's not a day that it's allowed to run on, it automatically moves forward by one day until it hits the next day that it is allowed to execute on.

Error codes:
PERMDENIED *box*
NOTBOOLEAN *field_value*
FIELDNOTEXIST *field_name*
INVALIDVALUE *value*
INVALID_DURATION *sched*

---

* schedule_get [-*box*] *sched_num* [*array_name*]

Description

Retrieves the indicated schedule for the current or indicated mailbox number (must be a decendant). Returns the information in an associative array if a variable name is indicated, else as an a-list of parameter and value pairs as the value of the function.

Error codes:
PERMDENIED
INVALIDKEY *sched_num*
INVALIDVALUE *value*

---

schedule_delete [-*box*] *sched_num*

Description

Deletes the indicated schedule.

Error codes:
PERMDENIED
INVALIDKEY *sched_num*
INVALIDVALUE *value*

---

* schedule_list [-*box*] [*var*]

Description

Retrieves a list of all schedules for the current or indicated user id. Returns the list as the value of the function unless *var* is specified, in which case that variable's value is set to the list and nothing is returned.

Error codes:
INVALIDVALUE *value*

`schedule_copy` *FromBox ToBox* `[-no_notify]`

Description

Copies all the schedules from one box to another box. This will ignore existing schedules in the *ToBox.* The *FromBox* can be any valid mailbox number, while the *ToBox* must be the logged in box or a child of the logged in box. Specify `-no_notify` to suppress notifying of the new schedules; this is useful for faster processing when creating *ToBox.* If *ToBox* specifies the currently logged-in mailbox, then that box must have its schedule lock attribute off to do any copying—that is, it must have the privilege to set its own schedules.

Error codes:
`PERMDENIED`
`INVALIDVALUE` *value*

12.2 Notifications and the Job Queue

Notify records are fired off whenever some predefined event happens, usually activity with your messages (receiving a new one, listening to all new ones, etc.). They can be used, for instance, to control a message waiting light on your phone, page a user when he gets messages, or to warn the system administrator about the disk space getting low so the he or she can assign more disk space to MMOs.

The notification subsystem consists of three different databases. The first, called *notification records,* describes, on a per-mailbox basis, the conditions under which the user wishes some command to be executed to accomplish some type of notification. For example, the user may wish to be paged whenever he receives voice mail which has been marked urgent.

Once the conditions of a notification record have been met, the system will automatically submit a job to the *job queue.* For historical reasons, the job queue is known as the *notify instance* database, since it holds "instances" of notification records which have "fired." However, in practice the notification instance records can created directly. For instance, an outbound calling application can submit job requests directly into this queue for processing, so it is also called the *outbound job queue,* or simply the *job queue.* The job queue automatically manages system resources to accomplish all of the requested work in an optimal fashion. For instance, if five hundred outbound calling jobs are submitted simultaneously, and there are only five outbound ports available, then the notify instance subsystem will create five threads of control which will execute the jobs one by one until all have been accomplished.

The second important aspect of the job queue is that it can repeat a job more than once on an automatic basis. Frequently, you will want a job to be repeated because it "fails." For instance, if its purpose is to call someone to notify them by voice of a waiting message, and it hits a busy signal, then the job would be considered to have failed to complete, and you'd want the job to be repeated after some interval until it succeeds. You can therefore specify the repeat interval, and the maximum number of successful or unsuccessful repetitions that should be performed before the job is deleted from the job queue.

Finally, the job queue can also manage the outbound calling operations so that they are executed only within restricted time periods. You can specify this by the day of the week, the time of the day, and/or the date. When a job would otherwise be executed but it is restricted from doing so by the day, date, and/or time, then it is automatically rescheduled to run at the first opportunity in the future when all the conditions for execution will be met. For instance, if a job is set to run only on weekdays at 11AM and to repeat once a day, then after its execution on Friday, the next execution will automatically skip forward to Monday since it is restricted from executing on Saturday and Sunday.

The third database is called the *notify template* database. Each job in the job queue must reference a notify template name which specifies *what* the job is supposed to do when it runs. These templates are simply Tcl procedures which all have a standardized proc name and argument list.

The commands which are used to access these three databases are as follows:

12.2.1 The Notify Record Database

`nfr_add [-`*box*`] `*type_list guard template variable successes failures days from to after every*` [`*port_group*`]`

Description

This function defines a new notify record for the current mailbox or for *box,* with the indicated field values. A user may create/delete his own and his children's notify records only if the mailbox has the `CTRIGGER` ("conditional trigger") privilege. This function does **not** check for duplicate records. The *type_list* is a tcl list of any combination of the following system known values or a type of another name:

`normal` A normal notification record is fired whenever a new message is received by the mailbox, regardless of the urgency of the received message.

`urgent` An urgent notification record is fired only when an urgent message is received by the mailbox.

`pickup` A pickup notification record is fired when the number of new messages in the mailbox transitions from one to zero. This will normally be when the user listens to those messages and they are marked "heard," but it can also happen if all the unheard messages are deleted.

`relay_page` An relay page record is fired when a relay page message is received by the mailbox.

The function `nfr_apply` can be used to fire off any of the types not known to the system.

The *guard* is a Tcl boolean expression. If an empty string is specified, the guard defaults to the constant `true`. When the notify record would otherwise fire, the guard is evaluated and if false, then the firing is suppressed. This can be used, for example, to page someone only when the number of new messages exceeds some threshold.

The *variable* field specifies an argument which will be passed to the notify template. In practice, this value is often a phone number which the template is to dial, but it can be any value which is meaningful to the template.

The remaining parameters are all passed verbatim to the notify instance database when the notify record is fired. See the description below of the corresponding arguments to the `notify_schedule` command.

The notify instance is also tied to the notify record which started it by the message's number and by the notify record's id. When the message which caused a normal or urgent notify to fire is marked deleted or heard, then any remaining outstanding notify instances for that message will be deleted automatically by searching the notify instance database for the corresponding notify record id.

Return values:
The id of the newly-created record.

Error codes:
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`KEYNOTEXIST` *template*

`nfr_del -[`*box*`]` *id...*

Description

Deletes notify record(s) for the current mailbox or for the indicated *box* by id. The ids are returned by the `nfr_add` and `nfr_list` commands. If a specified *id* is invalid, then an appropriate error message is returned and processing stops (ie, any *id*'s after that one are not processed).

Return values:
Empty string.

Error codes:
`BOXNOTEXIST` *box*
`PERMDENIED` *box*
`IDNOTEXIST` *id*

`nfr_list [-`*box*`]`

Description

Returns a list of all notify records defined for the current mailbox or the indicated *box*. The first element of each pair in the list is the notify record id, the second is a sub-a-list of notify record values.

Return values:
Tcl a-list of notify record ids and record value a-lists.

Error codes:
`INVALIDVALUE` *value*

`nfr_apply` *type variable*

Description

Checks all the notify records of the current mailbox for a record of *type*. If there are any, then they are fired off being passed the *variable*.

Return values:
Empty string.

12.2.2 The Notify Instance (Job Queue) Database

`notify_schedule [-box` *box*`]` *template variable successes failures days from to after every msg_num nfr_recno type* [*port_group*]

Description

This command schedules a job for automatic processing, which is typically an outbound call. It is also used internally by the notification record subsystem described above. The arguments have the following meanings:

box The notify will be placed in the specified box. Must have the `NOTIFY_ANY_BOX` privilege.

template The template name specifies which notify template (see section 12.2.3) contains the Tcl code to be executed by this job.

variable The variable field is a free-form argument which is passed to the notify template as part of its argument. It is copied from the notify record in the case of a notify; otherwise, it can be any value which has meaning to the notify template's code.

successes This specifies the maximum number of times that the template should be executed successfuly. If this maximum is reached, the job is deleted. A value of zero means there is no limit.

failures This specifies the maximum number of times that the template is allowed to fail. If this maximum is reached, the job is deleted. A value of zero means there is no limit.

days This value specifies the days of the week, Sunday through Saturday, when the job is allowed to execute. It can be expressed either as a decimal binary number (127 would mean all days) or as a string of `T`s and `F`s. The default is `TTTTTTT`.

from This value specifies the earliest time of each allowed day that the job can be executed. It can be expressed in one of two forms: an integer number of seconds since midnight or in HH:MM format using a 24-hour clock.

to This value specifies, in the same format as *from*, the last time in the day when the job is allowed to execute each day.

after This integer value specifies the number of minutes which must pass from the time the job is submitted before it is first executed.

every This integer value specifies the number of minutes which separate each successful or failed execution. That is, the job is repeated every *every* minutes, except for the day of week and time of day restrictions. If a job would otherwise run but it is restricted from doing so, it is rescheduled automatically for the next time when it is allowed to run.

msg_num If this job was being submitted automatically by a `normal` or `urgent` notify record, then the number of the corresponding message will be filled in here. When that message is subsequently marked deleted or heard, any outstanding notify instances for it will be deleted automatically. For non-notify jobs, use `-1` so as not to conflict with any actual message number. Note that the *msg_num* value is set as a global variable called `msg_num` in the interpreter which is executing the template.

nfr_recno If this job was being submitted automatically by a notify record, then this value will "point back" to the notify record which was fired. For non-notify jobs, use `-1`.

type If this job was being submitted automatically by a notify record, then this value will be the type of notification that was being executed (`normal`, `urgent`, etc.). It is checked when notify instance records are automatically deleted when messages are picked up/deleted to insure that only the correct records are deleted. Otherwise, it is unused and it can be given any other value, including the empty string.

port_group This optional parameter specifies the port group which will be used by the notify template. If left blank, then when the time comes to execute this job, a separate thread of execution is used. Otherwise, the notify instance subsystem optimizes the number of threads based on the number of ports in each identified port group.

Return values:
Empty string.

Error codes:
NOTNONNEG *successes*
NOTNONNEG *failures*
NOTNONNEG *after*
NOTNONNEG *every*
NOTINTEGER *msg_num*
NOTINTEGER *nfr_recno*

---

* `notify_list` [*box*]

Description

Notify instances are permanent until they have been stopped by their maximum number of successes or failures, or they are deleted by the `notify_delete` function or by a message pickup (or the deletion of the owning mailbox). The `notify_list` command shows all outstanding jobs for the current box, or for *box* if it's specified.

Return values:
The return value is a possibly-empty Tcl list of data about each outstanding job. The job-specific data will be the ID number of the job, the time it will next execute (as Coordinated Universal Time), the template it will execute, the "variable" argument to be passed to that template, the number of successful executions so far, the maximum number permitted, the number of failed executions so far and its maximum permitted value, a boolean value specifying whether the job is currently executing, a boolean value specifying that the record was deleted while it was executing (so it is to be destroyed when its execution finishes), and the notify record number and type information which were supplied to the `notify_schedule` *command.*

Error codes:
INVALIDVALUE *value*

`notify_delete [-box` *box*`]` *id...*

Description

The `notify_delete` command can be used to delete one or more notify instances by their ID numbers (as returned by the `notify_list` command). Normally, it operates only on IDs of notify instances belonging to the logged-in box, but other boxes' instances may be deleted by specifying `-box`, as long as the logged-in box is an ancestor of *box* or the logged-in box has the `NOTIFY_ANY_BOX` privilege.

Return values:
Emtpy string.

Error codes:
`IDNOTEXIST` *id*
`PERMDENIED` *box*

12.2.3 The Notify Template Database

`template_set` *name body*

Description

The `template_set` command allows a user to define or redefine notify templates. There is no set limit on the number of templates, except that template names are constrained to be no more than 16 characters, case insensitive. Since template names are really filenames, the character set allowed is the same as that for filenames in Win32. If the *name* specified identifies a template which already exists, that template is redefined. Otherwise, a new template is created. The *body* is the body of the `notify` procedure.

When a template is executed by the notify instance subsystem, it is passed a single argument. That argument will be the "variable" value which was specified to `notify_schedule`. When a notify instance is created by a notify record, the ID of the message corresponding to *msg_num* is appended to the variable field of the notify record; in this case, the list will contain two items, or possibly more if the "variable" field of the notify record was itself a list. The formal parameter which receives this single argument is called `notify_args`.

In addition to the argument, several variables are set globally in the interpreter when the notify proc is executed:

`msg_num` will contain the *msg_num* value that was passed to `notify_schedule`. This is used indirectly to implement the `%R` token.

`variable` will contain the value of the *variable* argument to `notify_schedule`. This is used to implement the `%V` token.

`last_failure` will be set to a Tcl boolean value. If true, then if the notify fails this time, it will not be rescheduled to execute again. If false, then an error return will cause the instance to be requeued. This information may be of use to the notify template so that it can clean up state (as in the MMO lookup table) if it is not going to execute again.

The notify template can also modify the value of `last_failure`. If it is set to true and the template returns an error, then the job will not be requeued again regardless of the number of failures so far.

`last_success` will be set to a Tcl boolean value. If true, then if the notify succeeds this time, it will not be rescheduled to execute again. If false, then a normal return will cause the instance to be requeued. This information may be of use to the notify template so that it can clean up state (as in the MMO lookup table) if it is not going to execute again.

The notify template can also modify the value of `last_success`. If it is set to true and the template returns normally, then the job will not be requeued again regardless of the number of successes so far.

The template body procedure should return an error, via the `error` command, if its execution is considered to have failed. Any normal return will be considered to be a successful execution. Not that the `tokens` command returns true if it completes the token execution and otherwise false; these return values may need to be translated to normal/error returns in the template body if it uses the tokens command.

This command is defined only if the logged-in user has the `CTRIGGER_CODE` privilege.

Return values:
Empty string.

---

* `template_list`

Description

This command can be used to enumerate the existing notify templates.

Return values:
A Tcl list of all defined template names.

---

* `template_get` *template_name*

Description

The `template_get` command loads *template_name* into the calling interpreter. This results in defining a function called `notify` with one argument, as described above.

Return values:
Empty string, but upon success it will have created or changed the `notify` *proc in the interpreter.*

Error codes:
`INVALIDKEY` *template_name*

`template_body` *template_name*

Description

The `template_body` command returns the body of *template_name*. This command is defined only if the logged-in user has the `CTRIGGER_CODE` privilege.

Return values:
The body element of the template.

Error codes:
`INVALIDKEY` *template_name*

12.3 Data Triggers

Amanda clients connect to the Amanda Portal server and allow you to send and receive voice mail through your computer rather than through your telephone. These clients also monitor different kinds of status information in the server. Whenever some data changes on the server, these clients need to be updated with the changes. These changes are performed through "monitors."

In Amanda Portal, there is code attached to the database modification functions that allow triggers to be set whenever data in the database is modified. These triggers normally send updates to the clients when the data changes.[1] To get this information, clients must register an "interest" in the data or they won't get the updates. This limits the amount of information that must be sent down to the client on a database change.

When a client connects, the client talks to a Tcl interpreter on the server just like a telnet session would, except that the server is actually talking to a client program and not a human. The client logs in and executes commands just as a human would.

Like the server, the client has a Tcl interpreter built into it and the server sends updates to the client as Tcl scripts. The updates are sent on a different port than the client sends requests to the server on so as not to confuse return results from client to server requests with updates from the server to the client. Figure 12.1 illustrates this.

This example shows a client asking for changes to box settings to be relayed to the client on port 5000. The client listens on port 5000 and the server connects to it. This dtrigger is assigned id "`id1`" which the client can later use to reference this dtrigger. Later, the client receives an update for a box setting on port 5000. Next, the client requests information about lmapping and autoschedule record changes to be sent on port 3000. Lastly, the client decides to no longer monitor box setting updates and gives the server the dtrigger id to delete. The server notices that this is the last dtrigger on that port and closes the port. When a client disconnects, the server simply deletes all the dtriggers that client was using.

In practice, the client doesn't ask for information to be sent back on multiple ports. One will suffice. Also, when you ask for information, you normally ask for information about a specific box or set of boxes, not all the boxes in the system.

Here is the call the client executes in the server to register interest in data changes. How the client sends this command to the server is not specified.

The dtrigger and monitor commands provided by the system exist as several different levels. The Amanda core provides three functions: `monitor_add`, `monitor_list`, and `monitor_delete`. The Internet DLL provides the functions `add_dtrigger` and `delete_dtrigger` in interpreters which are created for non-interactive (client program) connections only. These two functions are front-ends for the `monitor_add` and `monitor_delete` functions of the core.

[1] Currently, these changes are sent down the TCP/IP connection to the client; however, these triggers are more general than this. They actually call a callback in the Internet DLL that sends the data down the wire. Different DLLs could have different callbacks and do different things.

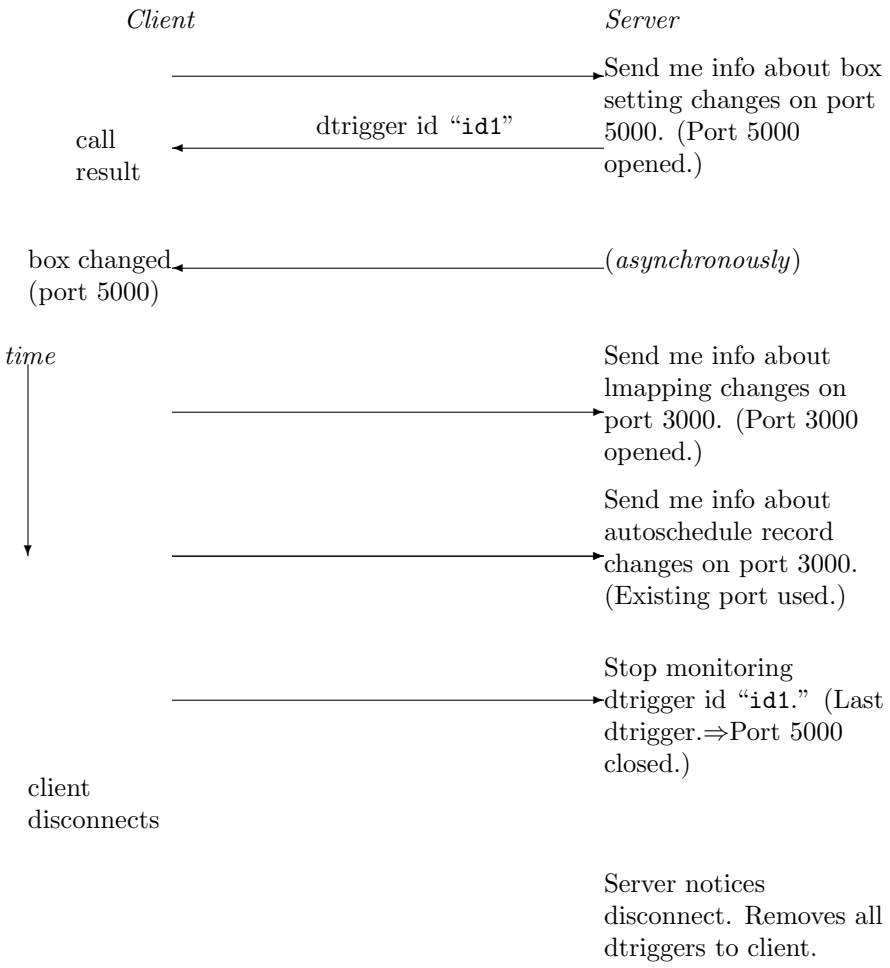

Figure 12.1: Example use of dtriggers.

Another command, `invoke_dtrigger` is called when the trigger is actually fired. It performs the actual work of contacting the client and delivering the information about the event which happened. Two other functions are used only for monitoring the system (observing port changes, capturing trace information, etc.). The reason they are separate is that they must be at a very low level in the system to avoid "feedback loops" such as tracing a trace event's processing. These functions are `add_trigger_request` and `delete_trigger_request`.

Finally, the Internet module also provides two additional functions for non-interactive (client) connections only: `internet_play` and `internet_record`. These two functions are substitutes for the `play` and `record` VP functions. Instead of playing/recording through a telephone interface, they establish TCP connections to the client at a predetermined port number, transfer some header information regarding the format of the file to be transferred, and then download or upload an MMO's content.

Here are the details of each of these functions:

`monitor_add` *type sync_group_id method cookie [ObjectName] [* `-permanent` *]*

Description

This function is used to create a new *monitor:* that is, a registration of interest by the client in some particular type of event. If successful, the monitor *id* of the newly created monitor will be returned.

The *type* argument must be one of the following strings: `shutdown`, `lmapping`, `tmapping`, `message`, `box_settings`, `box_creation`, `ttrigger`, `ctrigger`, `ctrigger_proc`, `mmo_lookup`, `call_queue`, `call_queue_view`, `persistent_proc`, or `privilege`.

When a notification to the client occurrs for the first time, it causes a TCP connection to be established to the client. Thereafter, for efficiency, the connection stays around until that monitor is deleted (or the client disconnects or logs out). We want to make sure that the client cannot be spoofed by a hacker, so it chooses a random value called a *cookie* which will be known only to it and the server. When a connection is made from the server to the client, the server must first send on that connection whatever *cookie* value the client had picked. If the client receives a connection but doesn't get the right cookie value from it, then it knows to reject that connection.

The *sync_group_id* argument is an arbitrary integer value that can be used by the client to group different categories of monitors. All monitors with the same *sync_group_id* will use the same TCP connection to the client for performing client updates.

The *method* argument is the Tcl command which should be run at the time that the monitored event happens. Normally this will be a form of the `invoke_dtrigger` command. This argument is normally formulated by the `add_dtrigger` command, since clients usually don't call the `monitor_add` funcition directly.

When the user logs out or calls `monitor_delete` then this notification is removed except if the `-permanent` flag was specified. If the `permanent` flag was specified then the notification will stay for the life of the process. This flag only works for the top level mailbox and does not apply to the messages and privileges notification triggers.

Return values:
id

Error codes:
`PERMDENIED` *ObjectName*
`HWRONGTYPE` *type*
`NOTINTEGER` *sync_group_id*

`monitor_list`

Description

Enumerates all the monitors that have been created by not deleted by this interpreter.

Return values:
Returns a Tcl list of lists. Each of the sublists is a triple of the id, ObjectName, and type, of the monitor.

`monitor_delete` *id*

Description

This function deletes a monitor by its *id*. The specified monitor must exist and have been created by this interpreter.

Return values:
A Tcl boolean value (0 or 1).

Error codes:
`NOTNONNEG` *id*
`IDNOTEXIST` *id*

`add_trigger_request` *port cookie type*

Description

This function adds a low-level monitor which will not cause further tracing or monitoring (so it is non-invasive, and it won't cause any infinite loops or deadlocks) within the system. It is used only for writing system monitoring programs.

The *port* argument is a TCP port number on the client to which the data should be sent. The *cookie* is a security cookie just like that in the `monitor_add` command described above.

The *type* argument must be one of the following: `port`, `user`, `trace`, `general`, `resource`, or `t1`. You may have at most one outstanding trigger request for each type.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *port*

`delete_trigger_request` *type*

Description

Deletes a previously-established low-level data trigger of type *type*. If no trigger of that type exists, no indication is given to the caller.

Return values:
Empty string.

`add_dtrigger` *type port cookie* [*box*]

Description

Tell the server to start sending changes related to *type* and *box* back to the client on port *port*. If *port* is not currently opened, it is opened up first. This command is only valid (and in fact only injected) in interpreters over a TCP/IP connection. You needn't give the IP address to send the data to because it can be figured out by inspecting the IP address the request came in on. To prevent trojan programs from sending data back on the port requested when the server calls back, a cookie is sent by the client. This cookie is returned by the server in all data transfers. If the client notices the cookies don't match, it closes the port because it isn't talking to the correct program. This security is rudimentary but sufficient for now. *type* can be one of the following:

`accept_call`: An incoming call is being presented to a call queue agent.

`box_settings`: Changes related to box settings.

`box_creation`: Boxes either being created or deleted.

`ttrigger`: The autoschedule records database being updated.

`ctrigger`: The ctrigger rule database being updated.

`ctrigger_proc`: The ctrigger_proc database being updated.

`lmapping`: The lmapping database being updated.

`tmapping`: The tmapping database being updated.

`persistent_proc`: The persistent procedure database being updated.

`call_queue`: The call queue being modified.

`call_queue_view`: The call queue updating active calls or active agents. Requires the view privilege on the queue.

`message`: Messages being added to, read or expunged in a box.

`privilege`: The privilege database being changed for a box.

`mmo_lookup`: An entry being changed in the MMO lookup table for a box.

`shutdown`: The server being shutdown.

All of the above *type*s except `box_creation`, `ctrigger_proc`, `tmapping`, `persistent_proc` and `shutdown` types require a *box* argument specifying the box to monitor.

Return values:
A dtrigger id that can be used to delete the dtrigger later.

Error codes:

errorCode	*Description*
`NOTNONNEG` *port*|*box*	
`BOXNOTEXIST` *box*	
`PORTPRIVILEGED` *port*	
`INVALIDTYPE` *type*	
`MISSINGPARAMETER` *box*	
`PERMDENIED`	Normally you can monitor the status of any box except with the `message` type. With this type, you may only monitor your own box.

errorCode *Description*

`delete_dtrigger` *dtrigger_id*

Description

Deletes the dtrigger identified by *dtrigger_id*. The information being sent by this dtrigger is no longer sent to the client. If this is the last dtrigger on the port, the port is closed.

Return values:
Empty string.

Error codes:
`IDNOTEXIST` *dtrigger_id*

`invoke_dtrigger`

Description

When the server sends data back to the client, it sends back a command like

type cookie dtrigger_id [*box*] *args...*

The *type* name is the same as the *type* name given in the `add_dtrigger` command. *That is, the client has a function with the same name as the type.* This function updates the client as appropriate. The *cookie* is sent back so the client can verify that it is actually the server that is sending it data. The *box* argument is only given for *type*s that are box specific. The rest of the *args* are *type* specific. They functions/data are defined as

---

`box_settings` *cookie dtrigger_id box* `set` *deleteability mutability* [*key value*]...

Setting *key* on box *box* has been changed to *value*. Each *key* has the *deleteability* and *mutability* given.

`box_settings` *cookie dtrigger_id box* `delete` [*key*]...

Setting *key* has been deleted on *box*.

`box_creation` *cookie dtrigger_id* `add`|`delete` *parent_box* [*child_box*]...

The child boxes given have been either created or deleted with relation to *parent_box*.

`ttrigger` *cookie dtrigger_id box* `add` *ttrigger_id* [*key value*]...

A new ttrigger for the specified box has been added. *ttrigger_id* is the id of the new ttrigger added.

`ttrigger` *cookie dtrigger_id box* `delete` [*ttrigger_id*]...

The *ttrigger_id*s given have been deleted from the system.

`ctrigger` *cookie dtrigger_id box* `add` *ctrigger_id* [*key value*]...

A new ctrigger for the specified box has been added to the system. The *key*/*value* pairs are the same as those given to `add_ctrigger`.

`ctrigger` *cookie dtrigger_id box* `delete` [*ctrigger_id*]...

The *ctrigger_id*s given have been deleted from the system.

`ctrigger_proc` *cookie dtrigger_id* `add`|`delete` [*proc*]...

The specified *proc*s have been added to or deleted from the ctrigger_proc database.

`lmapping` *cookie dtrigger_id box* `add`|`delete` *key* [*value*]...

Each of the *value*s have been added or deleted from the *key* for the specified box.

`lmapping` *cookie dtrigger_id box* `set` *key value_list*

The lmapping key has been set to the values in *value_list*.

`tmapping` *cookie dtrigger_id* `add`|`delete` *map_name* [*key value*]...

The set of values for the *key* given have been added to or deleted from the database.

`persistent_proc` *cookie dtrigger_id* `add`|`delete` [*proc*]...

The *proc*s given have been added to or deleted from the persistent procedure database.

`persistent_proc` *cookie dtrigger_id* `rename` *old_proc new_proc*

old_proc has been renamed to *new_proc* in the persistent method database. If *new_proc* is the empty string, the proc has been deleted.

`message` *cookie dtrigger_id box* `add` [*key value*]...

A new message has been added to *box*. The *key*/*value* pairs are the same as the array indices that are filled in by `get_next_msg` (see page 72) call except that the MMOs are not returned and the subject is not returned (which is also an MMO).

`message` *cookie dtrigger_id box* `add_fcc` [*key value*]...

A new message has been added as with *add*, except this is a filed carbon copy. The client may not wish to inform the user of new messages in this case.

`message` *cookie dtrigger_id box* `set` *msg_number* [*key value*]...

The message specified by *msg_number* has changed. It's new attributes are given by the *key/value* pairs. *msg_number* is the internal message number returned by `get_next_msg`. You can only monitor your own box with the `message` type.

`message` *cookie dtrigger_id box* `expunge` [*msg_number*]...

The messages specified by *msg_number* have been expunged.

`privilege` *cookie dtrigger_id box* `add` [*priv value*]...

Privileges have been added with the values given.

`privilege` *cookie dtrigger_id box* `delete` [*priv*]...

Privileges have been deleted.

`mmo_lookup` *cookie dtrigger_id box* `add` [*key a-list*]...

The keys given in the MMO database have been added or modified. *a-list* is an a-list containing the keys `description` and `private` as in the function `lookup_mmo_attrs` (see page 82). You will only get updates on private MMOs for your box only.

`mmo_lookup` *cookie dtrigger_id box* `delete` [*key*]...

The *key*s given has been deleted from the MMO Lookup Table for *box*.

`shutdown` *cookie dtrigger_id*

The system is shutting down. The system may take a while to shut down or may be shut down immediately. This call doesn't differentiate between the two.

`call_queue_view` *cookie dtrigger_id queue* `add call` *id* [*key value*]...

A new call, identified by *id*, has entered the *queue*. The call's attribute's are described by the key/value pairs. The key/value pairs have the same values as that given by the `call_info` command (see page 188).

`call_queue_view` *cookie dtrigger_id queue* `set call` *id* [*key value*]...

A call's attributes, identified by *id*, has changed as described by the key/value pairs. The key/value pairs have the same values as that given by the `call_info` command (see page 188).

`call_queue` *cookie dtrigger_id queue* `delete call` *id*

A call's has been deleted, or in other words removed from the queue, identified by *id*.

`call_queue_priv` *cookie dtrigger_id queue* `set accept` *id* `call` *callid* `status` *newstatus*

This is a notification of a call, *callid*, being assigned to an agent that has the option to accept or reject the call. New status of the agent is also updated.

`call_queue_priv` *cookie dtrigger_id queue* `set transfer` *id* `call` *callid* `status` *newstatus*

This is a notification of a call, *callid*, being immediatly transfered to an agent. New status of the agent is also updated.

`call_queue` *cookie dtrigger_id queue* `create queue` *box* [*key value*]...

A new queue has been created for *box*. The key/value pairs have the same values as that given by the `queue_info` command (see page 181).

`call_queue` *cookie dtrigger_id queue* `set queue` *box* [*key value*]...

A queue attributes has been changed. The key/value pairs have the same values as that given by the `queue_info` command (see page 181).

`call_queue` *cookie dtrigger_id queue* `expunge queue` *box*

The queue has been removed.

`call_queue_view` *cookie dtrigger_id queue* `create agentBox` *box* [*key value*]...

A new agent has been created for the *queue*. The key/value pairs have the same values as that given by the `agent_info` command (see page 182).

`call_queue_view` *cookie dtrigger_id queue* `set agentBox` *box* [*key value*]...

A agent's attributes has changed. The key/value pairs have the same values as that given by the `agent_info` command (see page 182).

`call_queue_view` *cookie dtrigger_id queue* `expunge agentBox` *box*

The agent has been removed from the *queue*. If *box* is -1 then all the agents have been removed from the *queue*, usually because the queue is being expunged also.

`call_queue_view` *cookie dtrigger_id queue* `add agent` *agentId* [*key value*]...

An agent has entered the *queue*. The key/value pairs have the same values as that given by the `agent_info` command (see page 182

`call_queue_priv` *cookie dtrigger_id queue* `set agent` *agentId* `status` *newstatus*

A agent's status has changed.

`call_queue_view` *cookie dtrigger_id queue* `delete agent` *agentId* `mailbox` *box*

The agent identified by *agentId* has left the *queue*.

`call_queue_stats` *cookie dtrigger_id queue* `set` *callStats* [*key value*]...

Real-time statistics update for the *queue*.

Chapter 13

Persistent Procedures

The system maintains an area on disk where you can store persistent Tcl procedures. Anybody with the EDIT_PMETHOD privilege (which will normally be limited to administrators) can use the `store_proc` command to store or modify a procedure in this persistent pool. Anybody can access and source the procedures in this persistent pool. You cannot store a procedure with the same name as another persistent procedure—that would redefine that produre to have the new arg list and body.

When you call a function in your Tcl script and that function isn't already defined as a command or a proc, the persistent procedure pool is searched for a procedure with a corresponding name, and if it is found, it is loaded into the interpreter and the command continues execution as if it had always been defined. Of course, once so referenced, the proc is now defined in the interpreter in case it is used again.

The persistent procedure pool mainly exists for the `CALLER_CODE`, `RNA_CODE`, *etc.*, settings on a box (see page 57). These settings consist of Tcl code to execute when a user calls in, when a user's extension is busy, *etc.* These fields are limited in length though, so they often simply contain calls to a persistent procedure. In addition, by factoring out the behavior from the setting on a box, we centralize the code into one place, making modification easier. If one or more functions are to made generally available, then they can be defined in a single file and that file can be added to the `tcl_source_files` configuration parameter so that those procs are defined in every interpreter. The persistent procedure pool's purpose is to hold procs that are seldom used, or are used by only one or a few mailboxes or under special circumstances, where it would be inefficient to define them in every interpreter that is created just on the off chance that it will be used.

The functions related to the persistent procedure pool are

`store_proc` *proc_name args body*

Description

Stores the procedure in the persistent procedure pool. You need the EDIT_PMETHOD privilege to execute this function.

Return values:
Empty string.

Error codes:
PERMDENIED
PROCNAMETOOLONG *limit proc_name*

---

* `proc_info body|args|procs` [*procedure*]

Description

Depending on the argument list, return different information. If `body` is specified, return the body of the procedure. If `args` is specified, return the argument list of the procedure. If `procs` is specified, return the name of all the procedures in the persistent procedure database.

The *procedure* argument is required if `body` or `args` is given. If `procs` is given, then *procedure* is optional and if not given, all the procedure names in the persistent procedure pool are returned. If it is given, then regular expressions determine which procedure names are returned, just like the "`info procs`" command.

The body argument can only be used if a logged-in user has the `EDIT_PMETHOD` privilege.

Return values:
Return value is described above.

Error codes:
PROCNOTEXIST *proc*
PERMDENIED

`rename_proc` *old_name new_name*

Description

Renames a persistent procedure. If *new_name* is the empty list, the persistent procedure is deleted. You need the EDIT_PMETHOD privilege to execute this command.

Return values:
Empty string.

Error codes:
PROCNOTEXIST *proc_name*
PROCNAMETOOLONG *limit proc_name*
PERMDENIED

Chapter 14

Integration

Integration and the need for it were discussed in Chapter 3. To simplify parsing integration information that comes back from the various phone switches, tAA came up with the concept of "patterns." Patterns are strings of characters—some characters with special meaning—that are matched with the information that came back from the phone switch. If a pattern matches, the information is extracted from phone switch data according to the pattern. This is very similar to the `scanf` strings in C.

Let's look at an example. Some switches give back information in prefix notation. That is, the data comes back like

reason arg...

For example, for a busy extension, the data may come back as

```
#A12
```

The "`#A`" says that the extension is busy and the "`12`" says that the extension that was busy is extension 12. We could create the pattern

```
#Abb
```

where "`bb`" has special meaning, in this case the extension that was busy. To handle a three digit extension, we can have another pattern like

```
#Abbb
```

We can create a number of these patterns and have a function that takes a list of patterns and the integration data and parses it out and returns the result. Since each switch may return the integration information differently, different sets of patterns are stored for different switches in the Configuration Database. Here is the definition of the command which performs the integration pattern parsing:

`parse_integration_data` *integration_data pattern_list*

Description

This function parses the integration data according to the pattern list supplied and returns the information found. The pattern matching algorithm is described below, along with detailed information about the format of the patterns.

Return values:

`NOMATCH`	The integration data didn't match any patterns.
`BUSY`	This call came to Amanda because and extension was busy.
`NOANSWER`	This call came to Amanda because an extension didn't answer within a set number of rings.
`RNA`	A call to an extension was ring-no-answer, based on the number of rings programming in the phone switch for that extension. If the call was from another extension and the switch supports it, `CallingMailbox` may also be supplied.
`TRUNK`	The call is an outside call. The trunk number will be stored in the variable `TrunkId` in the interpreter.
`LOGIN`	The call is a direct call from a station, which will be identified in the `CalledMailbox` variable. Normally the system will invite the caller to log into that box by asking only for a password.
`RECORD`	The call is an immediate record situation, with the mailbox identified in the `CalledMailbox` variable in the interpreter.
`HANGUP`	This was a hangup indication. Amanda usually ignores such calls.
`ANSWER`	This call type indicates that the switch detected an answer at a station. Amanda usually ignores such calls.
`CALLER_ID`	This is a call whose integration information specified only the Caller ID, and nothing else (such as an extension which was being called). Usually this will be a direct outside call. The Caller ID information will be stored in the `CallerId` variable in the interpreter.

Error codes:

`NOTLIST` *pattern_list*

The actual integration patterns will usually be defined in the Configuration Database in the PBX settings under the `integration` key. In integration patterns, the following letters (case sensitive) have special meanings to the `parse_integration_data` command:

All the DTMF digits 0–9, * and #, and A-D (represented as upper case letters) are *literals.* Where they occur in the pattern, the corresponding letter in the integration data must match. Finally, any other letters (conventionally, written as "x") are treated as wild-cards. They match any corresponding letter in the integration data. A pattern is considered not to match a given integration data string if their lengths are different. Once a match has been determined, then the positions in the patterns which contain the following letters will cause data to be extracted from the integration data and stored into variables in the interpreter. Finally, the presence of an a, b, e, h, i, or r in the matching pattern determines the overall type of call, which is returned by the parse_integration_data command described above.

a Answer state. This is usually discarded by Amanda.

b Busy state. Where "bbb" appears in the integration pattern is the extension which was busy. Sets the CalledMailbox variable in the interpreter.

c "Caller ID". This can actually be any data which indicates who the caller is. Sets the CallerId variable in the interpreter.

e Direct call from an extension "eee." Sets the CalledMailbox variable in the interpreter.

h Hangup. Amanda usually discards such calls.

i Immediate record. The extension "iii" wants to being recording the call that he is currently on, so Amanda should play the record tone and then start recording a message. Sets the CalledMailbox variable in the interpreter.

r Ring-no-answer. The extension "rrr" did not answer after some number of rings which is programmed into the phone switch (this is independent of the RNA field in the voice mailbox settings of Amanda). Sets the CalledMailbox variable in the interpreter.

s Station information. When Amanda receives a "bbb," "rrr," or "eee" call, the switch may also provide station information for who the caller was. This information will match the "sss" in the integration pattern. Sets the CallingMailbox variable in the interpreter.

t Trunk information. Amanda may use this information to process calls differently depending on which trunk group they came into the phone switch on (which will be different from which Amanda port group the call comes into from the switch).

When parse_integration_data matches an extension via "bbb," "rrr," etc., it must be translated into an Amanda mailbox number. In many situations, there is a one-to-one correspondence between these values, so no actual translation will occur. But in other situations, the extension number may not match the mailbox number. To handle this case, you may make entries in the Configuration Database's EXTENSION table for the extensions in question. The mailbox key tells the system to map that extension to the value provided with the mailbox key. So parse_integration_data looks in that database and tries to map the extension to a mailbox number. If no mapping is provided, then it is assumed that the mailbox number is the same as the extension number and no change is made to the data. Finally, parse_integration_data will set one or more variables in the interpreter to the possibly-translated mailbox numbers which matched the "bbb," "rrr," etc., patterns.

14.0.1 Serial Integration Modules

When Amanda Portal is connected to a phone switch which does serial integration, one of several support modules is loaded via the dlls Configuration Database parameter. These modules are currently SMDI, NECMCI, and GENERIC. When any of these modules have been loaded with the system, then the two commands described below become available.

Serial integration typically provides a two-way "conversation" between the voice mail system and the phone switch. The switch sends packets in some pre-defined format to Amanda to provide integration information about telephone calls that are coming to Amanda (or have just come). The switch, of course, describes the calls in terms of extension numbers on the switch which are connected to Amanda. So we need mechanisms to translate extesion numbers to Amanda port numbers and to mailbox numbers.

Secondly, serial integration often provides a way for the voice mail system to send status information to the phone switch. Most often, this is a way to control a message waiting indicator (MWI) on an extension, or equivalently, a stutter dialtone, which informs the user that they have unheard Amanda messages.

There may also be other aspects of the communication protocol, such as "heartbeat" checks, flow control issues, etc. All of these details are handled by the loadable DLL modules for the various protocols, and so they are invisible at the Tcl level.

---

* `get_serial_integration_data`

Description

This function returns the serial integration information for the call Amanda just received. This function can be called only from interpreters which were created in response to an incoming phone call. When integration information comes in, it may not come into Amanda in lock step with the call coming to Amanda. Therefore, Amanda has a window within which the integration information must be received before it is considered valid. This window is specified with the `smdi_pretimeout` and `smdi_timeout` values in the Configuration Database. If the serial integration information precedes the call, it is simply queued up and this function will quickly return it. Otherwise, this function will wait a short time until the information shows up.

Return values:

`NOINFO`	The serial integration information didn't show up or was outside of the timeout window.
Serial integration info converted to ASCII.	

`message_waiting_indicator` *extension setting*

Description

This function allows you to control the message waiting indicator status when a serial integration module is loaded. It is made available to all logged-in interpreters to that it can be executed by a notify job.

You specify the `extension` number and the *setting* of the MWI: `on` or `off`. This queues up a request to send out, on the serial connection, the appropriate sequence of characters to accomplish the requested change, and the `message_waiting_indicator` call returns immediately.

This function is defined only when the SMDI or NECMCI modules have been loaded with Amanda.

Return values:
Empty string.

Chapter 15

Call Queueing

The features and commands described in this chapter are available only if the CallQueue2 DLL is loaded with the system.

The identity of a user in Amanda Portal is closely associated with a box. You must log into a box to be known to the system. Each user (*i.e.*, box owner) may wish to have a queue of people waiting to call him. He would answer each call in turn and the people in the queue would wait until he is ready for the next call.

We can extend this concept by allowing other people the ability to receive calls destined for the owner of the box. In this way, we get a generalized queueing mechanism where queues are associated with boxes and we can allow other people to take calls destined originally for our box. Only one queue can be associated with each box in this manner though.

Call queueing gets slightly more complicated than this. You can actually use the queueing mechanism to support "network call screening" and "manage waiting calls." With "network call screening," you have a queue for your box but you are always unavailable. You select which phone calls you wish to take by looking at them on the screen and selecting the disposition of each call. You may wish to send a salesman's call to voice mail but take a call from your boss for example. With "manage waiting calls," you are actively taking calls but the calls queue up while you are talking. You can review the calls in the queue on your screen and decide to either preempt the call you are currently on to take another call in the queue, send one of the calls in the queue to voice mail or make the people in the queue wait. You can even set up rules on how to dispose of calls in the queue. You may wish to send certain calls automatically to voice mail based on their Caller ID information. This logic is not built into the system but rather programmable at the Tcl level.

Another usage of call queueing is to have a "hold queue." With a "hold queue" the owner of the queue does not actively monitor the queue. Calls get placed into the queue when the callee is busy on the phone. The calls get dequeued automatically after the caller at the top of the queue can successfully dial the callee's phone with an answer. Again, most of this logic is built into the Tcl level. The purpose of the call queue is to keep and order of the incoming calls for the callee.

Each box may or may not have a queue associated with it. Obviously, boxes that aren't created cannot have queues associated with them. The queue handle or identifier is the box number and we can create queues for specialized services, such as technical support, by creating a box for technical support and assigning a queue to it. To create a queue, you use the `create_queue` call. To do this, you will need the CREATE_QUEUE privilege. Once a queue is created, you can enable agents to attach to the queue to receive calls.

While we are limited to one queue to a box, a box owner can receive calls from multiple queues. When a call comes in, it is put in a call queue and an agent picks up the next call from the queue by signaling that he is ready to receive the next call. When the agent is done with the call, the system must be notified that the agent is done. If we are an unPBX (*i.e.*, there is no PBX. The computer transfers all the calls), then we know when the agent hangs up. If we are using a PBX, the agent must indicate to the system that he has hung up because the PBX doesn't notify Amanda Portal. The agent may do this by hitting a button on the screen of the Call Queue Agent application.

Calls can be inserted into a queue even if there are no agents attached to the queue. If there are no agents attached to a queue, the queue manager will act as a hold queue for the TUI to wait to be transfered to the mailbox. If there are agents attached to the queue, and the last one exits then the calls will get notified and will probably go to voice mail. (depending on the Tcl logic).[1] But if the call entered allowing no agents then it will not be notified when the last agent exits the queue.

Each agent that connects to a queue may have a number of "skills," such as speaking Spanish. Likewise, each call that comes in may require a number of "skills" of the agent that takes the call. If an agent is to take a call, they must have *at least* the skills required by the call. That is, if a call requires skills 'a' and 'b', then the agent must at least have the skills 'a' and 'b' to take the call. The agent may have more. When an agent is allowed into a queue, the queue manager tells the system which skills the agent possesses.

The rules for inserting calls into the queue and automatically removing calls from the queue are actually more restrictive because of skills. If a call has a set of skills and no agent is currently assigned to the queue that has those skills, then the call queue insertion function will fail. Also, if an agent leaves a queue and he is the last agent to fulfill a certain set of calls' skills criteria, then all those calls will automatically be sent to voice mail. (Actually the calls will be notified and Tcl will decide what to do with them.) So the rule given above about no agents in the queue, or the last agent's leaving the queue, are actually a special case of this more general rule.

[1] When you attach a call to a queue, you only get a variable handle that is used with the `wait` call to determine when your call queue status changes. The TUI Tcl code can interpret what to do when an event for this call is signaled.

When you add an agent to queue, you can specify additional "privileges" that that agent has. They are not privileges in the traditional global sense; rather, they are privileges that are specific to the queue model and are not set through the traditional privilege mechanism. You can specify

1. Whether the agent can "view" the calls and agents in the queue.
2. Whether the agent can reject a call being sent to him from the queue. This is known as the "rejection" privilege. In this case, the agent will get a popup on the screen asking him if he wishes to accept the call. He can accept or reject the call. If the agent waits too long to answer, a timeout occurs and the call goes back on the queue, but doesn't get transferred to that agent again.
3. Whether an agent can dispose of a call in an agent defined manner. That is, the agent can delete the call from the queue, send it to voice mail, take it out of order, *etc.* This "disposition" privilege implies the first one implicitly, since you couldn't grab a call from the queue and process it if you can't view the queue!
4. Whether an agent can change their state of availablity. If agent does not have this privilege then the agent is always avaiblable, unless taking a call.
5. Whether the agent is a "supervisor" of the queue. Normally as part of the global security model, in order to make changes you have to be either the owner or an ancestor with the right privileges. However with a call queue a supervisor can be set who is not an ancestor of the owner. This privilege allows the agent to do the same functions as the owner of the call queue. This implies that they have the "view" privilege, but not the "available" privilege. No calls can automatically be sent to a "supervisor", but if they have the "disposition" privilege then the "supervisor" can manually pick calls.

Figure 15.1 shows the behavior.

Call centers have a notion called "wrapup." When an agent is done with a call, the agent may have to finish up some paperwork related to the call. This is called the "wrapup" period. Some sites have a limit on the amount of time an agent can spend in the wrapup period. These sites will automatically send the next call in the queue to the agent if the agent exceeds the wrapup period timeout. If the "wrapup" period is zero then there is no "wrapup" period.

When the queueing code needs to transfer a call to an agent, it doesn't need to try out each agent's extension to see if that agent is busy. Instead, when an agent becomes available for the next call, he manually (except for unPBXes) indicates that he is available for the next call. When an agent connects to a queue, it is not assumed that that agent is available to take calls. He must indicate his availability first, unless the agent does not have the availablity privilege, in which case they are automatically available.

When a call is being transfered to an agent, the caller notifies the queue manager the success of the transfer. The queue manager can specify a timeout of how long the connecting process should be. If the timeout occurs or the call fails to be transfered then the call is placed back into the queue and the agent is placed into a temporary avoid list for the caller. The agent is also, placed unavailable, if they have the "available" privilege. This queue timeout is the `MAX_CONNECTING_SEC`.

A caller can exit the queue on their own via several ways. The caller can dial digits to go to another box or to go to voicemail. Before doing so, the caller can notify the queue manager why they are leaving. This is mainly for statistical reporting.

Once a caller and an agent is connected, the queue manager can set a minumum amount of time that they are in the connected state. The agent can not wrapup or change their availability state before this time. This queue timeout setting is the `MIN_CONNECT_SEC`.

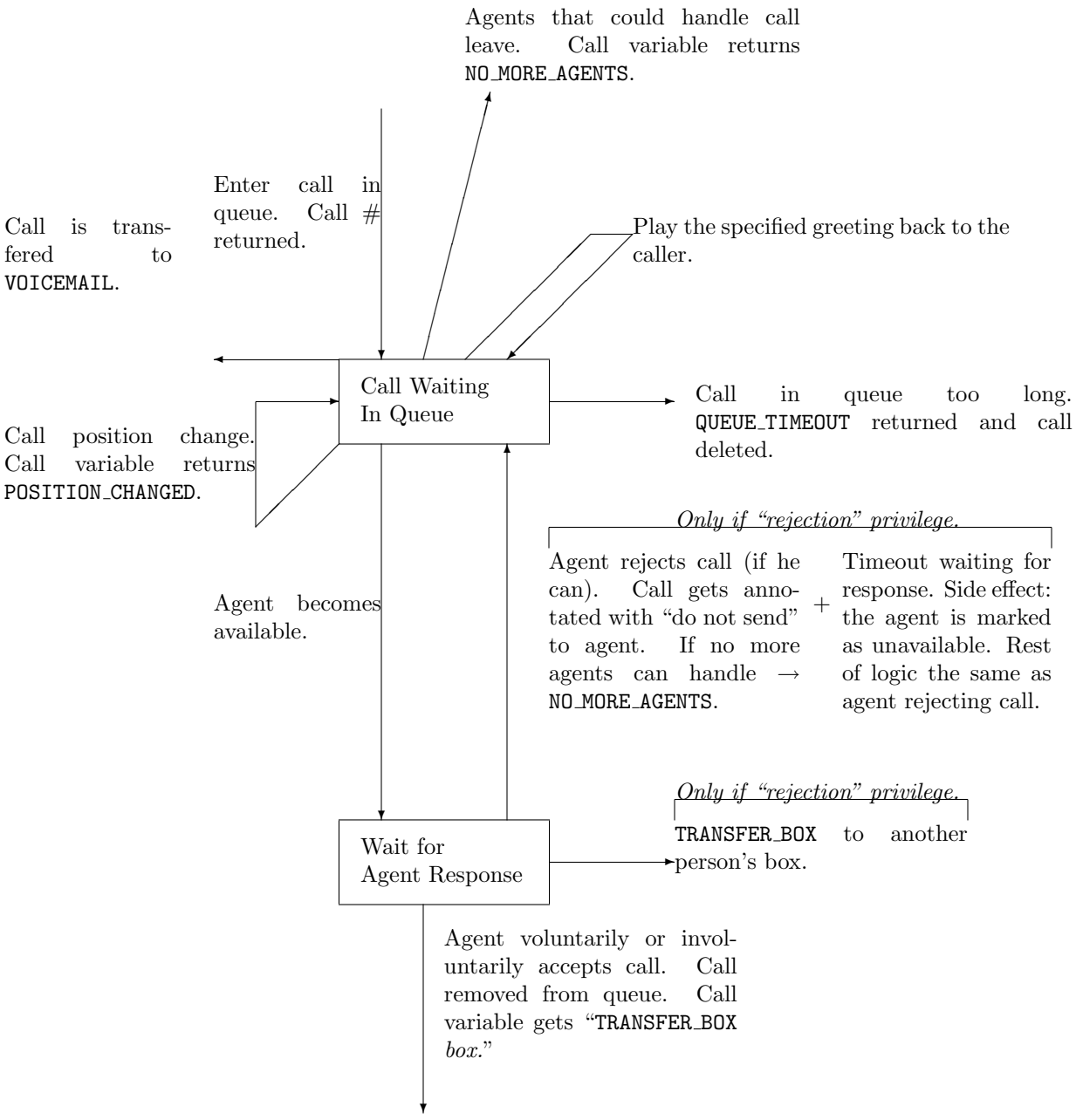

Figure 15.1: State Diagram for Call Queueing.

The queue manager can send updates to the callers about their position with in queue. There are three settings to choose from for the QUEUE_REPORT_POS: never, position, or time. If never is set then the callers will not get any updates. The callers can still ask for themselves where they are at in the queue. If position is set then the callers will be notified every time the callers position changes. If time is set then the callers will be notified when they first enter the queue how long they expect to wait before being connected to an agent.

The queue manager maintains a set of real time statistics in which it will continuesly notify all the agents of. One of the statistics is based on the queue setting QUEUE_HIGH_TIME. This is the number of seconds that a caller have been waiting that is deemed too long, but is still less than the ultimate queue timeout max_wait. The queue manager will report how many callers have reached this point. . When the TUI code adds a call to a queue, each thread gets back a handle that is useful with the wait command to determine when an event on a queue occurs. There are several types of events such as your position in the queue changes, an agent takes a call, *etc.* Here are the different events and their return values from the wait command. Refer to figure 15.1 for more information.

POSITION_CHANGED *pos [grt [grtBox]]*	The position of the call in the queue changed. The new position is the queue is returned. The call stays in the queue. Optionally a greeting to play is specified. The mailbox of the greeting is the queue mailbox unless *grtBox* is specified.
NO_MORE_AGENTS *[grt [grtBox]]*	There are no agents matching the call's skill set to handle the call. The call is removed from the queue. Optionally a greeting to play is specified. The mailbox of the greeting is the queue mailbox unless *grtBox* is specified.
QUEUE_TIMEOUT *[grt [grtBox]]*	The call was in the queue too long. That is, it exceeded the QUEUE_TIME which was specified by the creator of the queue (this may be used to set a policy that callers who wait longer than some given time are sent to voicemail for a later callback, for example). The call is removed from the queue. Optionally a greeting to play is specified. The mailbox of the greeting is the queue mailbox unless *grtBox* is specified.
TRANSFER_BOX *box [grt [grtBox]]*	Transfer the call to the indicated agent. If the agent has the ability to reject the call, then you will not get this event until the agent has actually decided to accept the call. See the state diagram. The call is removed from the queue. The event result does not include the extension to call the agent on. This mapping from box # to the agent's extension is outside of the domain of the call queueing logic. The agent may not even be at an extension on the PBX switch. The agent may be at home. When you get this result from wait, the target agent is automatically made unavailable. Optionally a greeting to play is specified. The mailbox of the greeting is the queue mailbox unless *grtBox* is specified.
PLAY_GRT *[grt [grtBox]]*	The call is stays in the queue. Optionally a greeting to play is specified. The mailbox of the greeting is the queue mailbox unless *grtBox* is specified.
VOICEMAIL *box [grt [grtBox]]*	Send the call to voice mail for the indicated box. The call is removed from the queue. Optionally a greeting to play is specified. The mailbox of the greeting is the queue mailbox unless *grtBox* is specified.
NOSKILLS_AVAIL *[grt [grtBox]]*	There are no agents in the queue that possess the skills required to take the call.

The functions for queue manipulation are as follows:

---

create_queue [-queue *queue*] [*extended_field*]...

Description

This function create a queue for the user's box or one of his descendent boxes if the `-queue` option is given. *queue* is a box number. You must have the CREATE_QUEUE privilege to create a queue for a box. *extended_field* is a list of extended fields that will be displayed on the screen when the calls are displayed. They are user defined.

Return values:
Empty string.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*
`QUEUE_EXISTS` *queue*
`INVALID_COLUMN` *column*
`COLUMN_EXISTS` *column*

`delete_queue` [`-queue` *queue*]

Description

This function deletes a queue from a box. You may not delete a queue if any agent is attached to it. The default queue is the current box if `-queue` is not given. You need the CREATE_QUEUE privilege to delete a queue for a box.

Return values:
Empty string.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*
`AGENT_ATTACHED` *queue*

`create_queue_columns` [`-queue` *queue*] [*key value*]...

Description

Create additional parameters to be used by the queue. The parameters are not necessarily known to the queue manager, but they can be utilized in the Tcl level logic. One example of this is setting up comfort messages that is played every so often to the caller. Once the parameters are created you then can use the `set_queue_setting` to set the values of the new parameters. You need the CREATE_QUEUE privilege to set settings on a queue.

Return values:
Empty string.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*
`INVALID_COLUMN` *column*

`set_queue_setting` [`-queue` *queue*] [*key value*]...

Description

Sets different settings for a queue. You need the CREATE_QUEUE privilege to set settings on a queue. The extended fields of the *queue* can be set along with the known settings as follows:

`max_wait` *seconds*	If a call sits in the queue longer than *seconds*, it is automatically deleted from the queue and `QUEUE_TIMEOUT` is returned by the `wait` call. You can pass the string "`0`" for *seconds* if you want an infinite timeout. This is the default.
`max_calls` *size*	If the queue reaches this size, any `enqueue_call` call will fail with `MAX_QCALLS`. The default is no limit on the size of the queue.
`require_wrapup` *boolean*	Allow a wrapup period after taking a call. The default is `false`.

Return values:
Empty string.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*
`INVALIDVALUE` *key value*

`get_queue_setting` [`-queue` *queue*]

Description

Gets different settings from a queue. The extended fields of the *queue* are, also, retrieved along with the known settings. This will return a a-list of the settings as described in set_queue_setting.

Return values:
a-list.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*

`enable_agent` [`-queue` *queue*] [*agent_box*]...

Description

This function adds the users of each *agent_box* to the list of boxes that are allowed to handle calls for the box currently logged in. You need the CREATE_QUEUE privilege to set settings on a queue. It does not indicate that the *agent_boxes* should start receiving calls for the queue, just that they are allowed to receive calls if they wish. An agent must then call `attach_to_queue` to actually start receiving calls. If `-queue` is given, then modify the given *queue* rather than the current box. You need to call `set_agent_settings` to modify the agent's per queue privileges.

Return values:
Empty string.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*
`INVALID_AGENT` *queue*
`AGENT_EXISTS` *agent*
`BOXNOTEXIST` *agent*

`set_agent_setting` [`-queue` *queue*] *agent_list* [*key value*]...

Description

This function sets different settings for a list of agents on a per queue basis. You need the CREATE_QUEUE privilege to set settings on a queue. The settings are

`wrapup` *seconds*	Set the wrapup time in seconds. If set to zero, there is no wrapup time. The default is infinite amount of wrapup time. If an agent wants to only receive calls at their discretion, they can either set themselves unavailable and then make themselves available when they want to receive the next call or if they have the "disposition" privilege, they can take calls from the queue even when they are unavailable.
`rejection_timeout` *seconds*	Agents who can reject calls have a limited time to accept or reject them when the call is sent to them. If they exceed this timeout, the call is put back on the queue and marked so not to be resent to that agent. The default for this value is 10 seconds. You can give the string "`0`" here for an infinite timeout, but this is *not* recommended.
`view` *boolean*	Enable or disable the "view" privilege (see below).
`rejection` *boolean*	Enable or disable the "rejection" privilege (see below).
`disposition` *boolean*	Enable or disable the "disposition" privilege (see below).
`skills` *skill_list*	A list of skills that the agent can handle.
`available` *boolean*	Enable or disable the "available" privilege.
`supervisor` *boolean*	Enable or disable the supervisory position of the agent to the queue.

There are several privileges each agent can have:

1. The agent can view the calls and agents in the queue (`view`).
2. The agent can reject a call being sent to him from the queue (`rejection`). In this case the agent will get a popup on the screen asking him if he wishes to accept the call. He can then accept or reject every call as it comes to him. The agent should use the `accept_call` member function to accept or reject a call.
3. The agent can dispose of a call in a way he sees fit (`disposition`). This privilege implies the first privilege implicitly.
4. The agent can change their own state of availability (`available`). If they do not have this privilege then the agent is always available unless in a process of a call.
5. The agent be a supervisor of the queue. (`supervisor`). If they have this privilege then they can act as the owner of the queue. They are alway unavailable if this privilege is enabled.

By default, all the privileges are disabled. You can enable the privileges with the above settings.

Return values:
Empty string.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*
`INVALID_AGENT` *queue*

`enumerate_agents` [`-queue` *queue*]

Description

This function returns a (possibly empty) list of all the agents (mailboxes) which are enabled as agents for the specified queue. If *queue* is not specified, then the current mailbox's queue is assumed. If no agents are enabled for the queue, then an empty list is returned.

Return values:
Possibly empty Tcl list of the agents for the specified queue.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*

`get_agent_setting` [`-queue` *queue*] [*agent*]...

Description

This function returns agents enabled on a queue and the settings for each of those agents.

Return values:
The return value is an a-list of a-list. The first a-list is indexed by the agent mailbox number. The second a-list is indexed by the settings. The settings are the same as in `set_agent_settings`*. You don't need any privilege to youself as an agent of the queue, but to get the settings of other agents you do need the privilege.*

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*
`INVALID_AGENT` *queue*

`disable_agent` [`-queue` *queue*] [*agent*]...

Description

This function disables the *agent*s from the list of boxes allowed to receive calls from the queue. If any of the *agent*s are currently talking to a call from the queue, the call is not killed and the variable the agent is using is still valid and the agent can still receive calls. He will not be disabled until he actually tries to attach again. This is much like the open file semantics in Unix. If `-queue` is given, then modify *queue* rather than the current box's queue. If an agent is not currently enabled on a queue and you try to disable him, no error is given. You need the CREATE_QUEUE privilege to disable agents on a queue.

Return values:
Empty string.

Error codes:
`PERMDENIED` *queue*
`INVALID_QUEUE` *queue*
`INVALID_AGENT` *queue*

The following commands fall into two categories: those intended for agents (listed first) and those intended for callers. Both create a handle in the form of a Tcl variable. For clarity, the agent's handle on the queue will be called *qhandle,* while a caller's handle will be called *chandle.*

`attach_to_queue` *queue qhandle*

Description

This function attaches an agent to a queue. You must give the queue. There is no default because you often will attach to a box that is not your own box. *var* is a handle to your attachment. It is a variable referencing a member function much like MMO handles. The calls you can use with this handle are described below. If you unset the handle, you are removed as an agent for the queue. Initially, when you attach to a queue, you are listed as `unavailable`, unless you don't have the available privilege, in which your state will be `available`. You must call the `available` or `wrapup` functions before you are considered available to take a call. You cannot attach to a queue if you are already attached. You will get an error. Any existing *var* is silently replaced.

Return values:
agent_id.

Error codes:
`INVALID_AGENT` *agent*
`QUEUE_MAN_EXISTS` *queue*
`MULT_ATTACH` *agent*
`QUEUE_DB_DELETED` *queue*

`enqueue_priv` *queue privilege_name*

Description

This returns back `1` if the current mailbox has the *privilege_name* within the specified *queue*. Otherwise, `0` is returned.

Return values:
privilege_value.

The member functions for agents on queues are as follows. Each command has a `cmds` member function so you can use `make_local` command with it if you wish.

---

qhandle `dispose` *call_id* `transfer|voicemail|hold|allow_timeout` [[`-mailbox` *mailbox*] — [`-agent` *agent_id*]] [`-greeting` *grt [grtbox]*]

Description

Disposes of a call specified by *call_id*, as the agents sees fit. The agent must have the `disposition` privilege to use this command. The agent can either `transfer`, `voicemail`, or `hold` the call to either a *mailbox* or *agent_id*. If `greeting` is specified then *grt* of either the queue mailbox or if specified the *grtbox* is played back to the caller. If the disposition is `voicemail` then the call is removed from the queue. If the disposition is `transfer` to a *mailbox* then the call is removed from the queue. If the disposition is `transfer` to a *agent_id* then the call remains under the management of the queue. If the disposition is `hold` then *mailbox* and *agent_id* options are ignored and the call remains in the queue. If the disposition is `allow_timeout` then the the *grt* value specifies if the queue's setting *max_wait* will take affect or not. This is useful in keeping a caller in the queue beyond the *max_wait* setting.

Return values:
Empty string.

Error codes:
`PERMDENIED` *agent*
`INVALIDVALUE` *grt*
`INVALIDVALUE` *call_id*
`INVALIDVALUE` *agent_id*
`BOXNOTEXIST` *grtbox*
`BOXNOTEXIST` *mailbox*
`INVALID_AGENT` *agent*

---

qhandle `stop`

Description

Stops the waiting on the *qhandle*.

Return values:
Empty string.

---

qhandle `event`

Description

Returns the event handle to wait on the *qhandle*.

Return values:
event.

---

chandle `result` [`-nowait`]

Description

Returns the result of the implicit wait operation. When this agent's status changes, then the result will become available. If `nowait` is specified then this will return immediately, otherwise this agent will wait until a result is sent.

---

qhandle `available` [`-agent` *id*] *boolean*

Description

Tells the system that the agent is available or unavailable to take the next call. Requires the `available` privilege.

If *boolean* is true, then the agent is available to take the next call immediately. When true, this function serves a dual purpose. It notifies the system that the agent is ready to take a call immediately (perhaps when the agent is done with wrapup before the wrapup timeout) and it notifies the system that the agent is no longer unavailable.

If *boolean* is false, then the agent is unavailable. An agent may make himself unavailable for a short period, such as time to use the bathroom, or a long period of time, such as when an individual is using these features to effect "network call screening." An agent could make himself unavailable just by removing himself from the queue but then the handle would disappear and any calls that only matched that agent would be removed. These calls currently stay in the queue until the agent makes himself available again.

To specify the `agent` option, you must have the `supervisor` privilege. This option allows the supervisor to set the availability of any of the attached agents.

Return values:
Empty string.

Error codes:
`PERMDENIED` *agent*
`STATUS_ERROR`

---

qhandle `wrapup` [*stat_data*]

Description

Tells the system that the agent is starting wrapup mode. If the wrapup timer is set to zero, he will get the next call immediately; otherwise, he will get the next call after the wrapup timer expires. If you want to be available before the wrapup timer expires, call "*qhandle* `available true`." If *stat_data* is specified then that infomation is recorded in the statistical database about that call.

Return values:
Empty string.

Error codes:
`STATUS_ERROR`

---

qhandle `accept_call` *boolean*

Description

Accept or reject a call being sent to you. You must have the "rejection" privilege to use this call. The agent is notified of a potential incoming call with a the `accept_call` dtrigger described on page 154. After receipt of the dtrigger, a dialog box is posted on the screen and the agent can accept or reject the incoming call using this member function. Notice that you needn't give the call number. The system knows which call it is trying to send you. An agent can only get sent one call at a time regardless of the number of queues he is attached to.

When the agent gets sent a call for acceptance or rejection, he is automatically marked for deciding call. If he rejects the call, he is marked back to the agent's previous state whether it was `available` or `unavailable`, but the call is annotated with information noting that the call should not be sent back to the same agent again (see figure 15.1). This is also true if the agent doesn't respond within the `rejection_timeout`. If the agent accepts the call, he is marked unavailable and the call is removed from the queue. The "`TRANSFER` *box*" result is returned by `wait` in this case.

Return values:
Empty string.

Error codes:

errorCode	*Description*
`PERMDENIED` *agent*	
`STATUS_ERROR`	
`INVALID_AGENT` *agent*	
`ACCEPT_STATUS_ERROR`	

---

qhandle `queue_info`

Description

This returns information pertaining to the queue. You *do not* need the "view" privilege to execute this call.

Return values:
An a-list with the following key/value pairs:

`max_wait` *seconds*	Queue timeout.
`avg_wait_time` *seconds*	Average time a call is waiting in the queue.
`wait_time_per_call` *seconds*	Average time a call is waiting in the queue per position change (dequeue rate).
`require_wrapup` *boolean*	Whether or not the queue requires a wrapup of the call.
`extended_fields` *extended_fields*	This is a list of extension fields that were given when the queue was created. If there were no extension fields given, the value is an empty list.

---

qhandle `agent_info`

Description

This command returns all the information about the agents in a queue. You need the "view" privilege to execute this call. If you don't have the "view" privilege then you will only get information about yourself.

Return values:
The return value is an a-list of a-lists. The first a-list contains the agent_id for each agent in the queue. The value of the array at this index is another a-list that contains the following key/value pairs:

`agent` *id*	The current queue manager's id of the agent.
`mailbox` *box*	The mailbox of the agent.
`view` *boolean*	Whether the agent has the "view" privilege.
`rejection` *boolean*	Whether the agent has the "rejection" privilege.
`disposition` *boolean*	Whether the agent has the "disposition" privilege.
`available` *boolean*	Whether the agent has the "available" privilege.
`supervisor` *boolean*	Whether the agent has the "supervisor" privilege.
`status` *status*	The status of the agent. The status can be `deciding`, `available`, `unavailable`, `in_call`, `wrapup`, `init_in_call`, or `connecting`. See the `status` member function for the meaning of these states.
`skills` *skills*	A list of skills the agent possesses.
`wrapup` *seconds*	The agent's wrapup time.
`rejection_timeout` *seconds*	For agents with disposition privilege, the amount of time they have to decide what to do with the call.

Error codes:

errorCode	*Description*
`PERMDENIED` *agent*	

---

qhandle `position`

Description

This function returns the agent's position. As agents are assigned to calls, the positions of the remaining available agents are decremented until they reach position 0, at which point they are eligible to be assigned to a call. If an agent is not available to take a call then -1 is returned.

---

qhandle `call_info`

Description

This command returns all the information about the calls in a queue. You need the "view" privilege to execute this call.

Return values:
An a-list of a-lists. The first a-list is a list of the call_id. The second nested set of a-lists has the same key/value pairs as the `call_info` *command on page 188.*

---

qhandle `active_call_info`

Description

This command returns all the information about the call an agent is connected to in a queue. You don't need the "view" privilege to execute this call.

Return values:
An a-list of a-lists. The first a-list is a list of the call_id. The second nested set of a-lists has the same key/value pairs as the `call_info` *command on page 188.*

Error codes:

errorCode	*Description*
`INVALID_CALL` *agent*	

---

qhandle `status`

Description

Returns the current status of the agent corresponding to the handle.

Return values:

`available`	Agent is ready to connect to a call.
`unavailable`	Agent is temporarily unavailable to take a call.
`in_call`	Agent is currently connected to a call.
`wrapup`	Agent is doing wrapup.
`deciding`	Agent has the "rejection" privilege and has been notified of a new call but has not yet responded.

---

qhandle `stats`

Description

Returns the current statistics of the queue, since the queue manager started its session (not since inception.)

Return values:
The return value is an a-lists that contain the following key/value pairs:

`attached` *number*	The current number of attached agents.
`available` *number*	The current number of available agents.
`busy` *number*	The current number of agents busy working on a call.
`unavailable` *number*	The current number of attached unavailable agents.
`lost` *number*	The total number of calls that hungup.
`reload` *number*	The total number of calls that exited the queue by dialing digits.
`answered` *number*	The total number of calls that have been answered by agents.
`waiting` *number*	The total number of calls currently waiting in the queue.
`high_time` *number*	The total number of calls currently waiting in the queue for longer than `QUEUE_HIGH_TIME` seconds.
`calls_total` *number*	The total number of calls that have entered the queue.
`voicemail` *number*	The total number of calls that have gone to voicemail.
`hour_answered` *number*	The total number of calls that have been answered by agents within the last hour.
`hour_lost` *number*	The total number of calls that hungup within the last hour.
`hour_reload` *number*	The total number of calls that exited the queue by dialing digits within the last hour.
`hour_total` *number*	The total number of calls that have entered the queue within the last hour.
`hour_incoming_rate` *number*	The rate of calls coming in per hour.
`hour_answer_rate` *number*	The rate of calls being answered per hour.
`hour_voicemail` *number*	The total number of calls that have gone to voicemail within the last hour.

---

qhandle `call_report` [`-by` *hour/day/daily/week/month*] [`-from` *cut*] [`-to` *cut*]

Description

This function queries the call statistics for the associated queue. It filters the statistics to be only those which occured in the range of the `-from` to `-to` times. If `-from` is not specified, then the statistics database will be searched from the first record; if `-to` is not specified, then records up to the present time will be included. Calls which have not yet completed are not included in the results.

The `-by` option allows you to specify how the data should be grouped. The default is `-by hour`. Grouping the data `-by day` means by day of week, whereas `-by daily` means by the day of the month. Weeks are 7-day periods starting on a Sunday. The first and last weeks of a month can, of course, be less than 7 days long. Weeks are identified by the week number within the containing month.

---

qhandle `agent_report` [`-by` *hour/day/daily/week/month*] [`-from` *cut*] [`-to` *cut*] [`-agent_id`]

Description

Two different types of agent reports can be obtained from the queue statistics database. In both types, the `-from` and `-to` arguments can be used to specify the range of time over which the report should be run, with the defaults being to go from the first record in the statistics database up until the most recent.

The first is a report about a specific agent. In this type of report, the data is summarized by time periods similar to call reports. In this case, the `-by` argument can be used to specify which periods to summarize the data over, with the default being by hour.

The second type of agent report covers activities by all agents of the queue. In this case, the `-from` and `-to` arguments may be used, but the `-by` and *agent_id* arguments are not used. The function returns an a-list of agent_id values with statistical data about each agent as the associated value. The data summarizes that agent's activities over the entire period specified by the `-from` and `-to` arguments or their default values.

---

qhandle `cid_report` [`-from` *cut*] [`-to` *cut*] *phone_number*

Description

This command generates a Caller ID report for the queue. The `-from` and `-to` arguments can be used to specify the range of time over which the report should be run, with the defaults being to go from the first record in the statistics database up until the most recent.

The *phone_number* argument must be specified, and the database will be searched for all calls from that phone number during the specified time range. Any calls will be returned as an a-list with the key being the phone number and the attributes specifying when

---

To add a call to a queue you use the `enqueue_call` command. The primary thing that a caller does with the returned handle is to `wait` on it. Waiting on it returns one of the strings listed on page 172.

---

* `enqueue_call` [`-top`] [`-even_if_no_agents`] [`-only_if_queue`] *queue var* [*key value*]...

Description

Enqueue a call on a queue. You must give the queue number. A variable is returned representing the call on the queue. You can use the `wait` command to wait on this variable for queue status changes. The variable returned also has a member function binding like MMO handles. Initially, the call is set for `transfer` true (see below). The *a-list* is a list of attributes about the call. The attributes are as follows:

`skills` *skills*	Necessary skills needed for this call.
`port` *port*	The telephone port the call is on.
`caller_id` *phone_num*	The telephone number the caller is calling from.
. . . extended field values. . .	

Naturally, even logged-out callers can use this command, so it requires no privileges. To delete a call from a queue, unset the variable that is passed to `enqueue_call`.

The `-top` option inserts a call at the top of the queue. There is no privilege for doing this, so any call can insert themselves at the top of the queue. However, this is not a problem because callers cannot enter Tcl code over the phone—the are restricted to executing pre-existing code. When a call hits the top of the queue and is sent to the agent, the agent box number the call is sent to is recorded internally. If the agent doesn't answer, or for some other reason we need to requeue the call, then we can use the `-top` option so that the call isn't put back at the end of the queue after having waited to reach the top already. When you use this option, it has the side effect that the agent the call was originally sent to is made unavailable as if the agent had executed the "*qhandle* `available false`" call. Since the agent didn't answer, it is assumed he is unavailable but forgot to inform the Amanda system of this fact.

The `-even_if_no_agents` option allows the call to enter a queue even when there are no agents attached to the queue. This is useful for a simple holding queue.

The `-only_if_queue` option allows the call to enter the queue only if the queue is already in place.

Return values:

Call id for call.	The call id is returned.
`AGENTNOTAVAIL`	Agent not available. This can be because there are currently no agents monitoring the queue or there are no agents with the skill set required monitoring the queue. This result can also be returned because the indicated queue does not exist. Either way, the call cannot wait in the requested queue.
`QUEUEFULL`	`max_calls` is set and the queue is full. No more calls can be inserted into the queue.
`NOSKILLS_AVAIL`	The queue does not have agents attached with matching skills.
`QUEUE_EMPTY`	The queue does not have agents attached.

Error codes:

errorCode	*Description*
`INVALID_QUEUE` *queue*	
`ALREADY_ENQUEUE` *queue*	
`MAX_QUEUE_ENTERED` *queue*	

---

* `agent_available` *queue*

Description

This tests the *queue* if there are any agents currently attached.

Return values:
0 if no agents, 1 if agents attached

---

chandle `transferable` *boolean*

Description

Determines whether a call will be transferred to an agent when it reaches the top of the queue. If *boolean* is true, the call will be transferred. If *boolean* is false, it will not be transferred. Setting this to false also prevents an agent from disposing of the call in any way if the agent has the "disposition" privilege. That is, the agent can't take the call, can't delete it, *etc.* It is the caller equivalent of an agent's being available or not.

The primary purpose of this function is to allow a caller to use IVR or some other audio application while waiting in the queue. Perhaps they are allowed to listen to menus which answer commonly-asked questions on the hope that they will answer their own question while still in the queue. While they are doing this, they are not monitoring their queue variable, so they need to indicate that they are not available to be transferred to an agent for that time period.

If all agents that can handle a call leave the queue, this call is dispositioned with a `NO_MORE_AGENTS` response no matter what transfer is set to.

Return values:

`CALL_TRANSFER`	The call can't change the transfer state.

---

chandle `position`

Description

Returns the position of the call in the queue.

Return values:
The queue position of the call.

---

chandle `queue_info`

Description

This returns information pertaining to the queue as a whole.

Return values:
An a-list with the following key/value pairs:

`max_wait` *seconds*	Queue timeout.
`avg_wait_time` *seconds*	Average time a call is waiting in the queue.
`wait_time_per_call` *seconds*	Average time a call is waiting in the queue per position change (dequeue rate).
`require_wrapup` *boolean*	Whether or not the queue requires a wrapup of the call.
extended_fields extended_fields	This is a list of extension fields that were given when the queue was created. If there were no extension fields given, the value is an empty list.

---

chandle `call_info`

Description

This command returns all the information about the call in the queue.

Return values:
An a-list. The a-list has the following key/value pairs:

`skills` *skills*	Necessary skills needed for this call.
`caller_id` *phone_number*	Phone number of caller.
`call_state` *state*	*state* is either `enabled`, `disabled`, `wait_for_agent_response`, `call_taken`, `connecting`, or `transfer_failed`. When `enabled`, the call is allowed to be transferred to an agent. When `disabled`, a `transferable false` has been executed and the call is not going to get sent to an agent.
`time_entered` *secs*	Time call entered queue in number of seconds since 1970.
`position` *n*	The position of the call in the queue.
`agent` *id*	The agent id assigned to the call
`agentBox` *box*	The agent mailbox assigned to the call
. . . extendend field values. . .	

---

chandle `status`

Description

Returns the current status of the call.

Return values:
Status can be one of the following values: `enabled`*,* `disabled`*,* `wait_for_agent_response`*,* `connecting`*,* `call_taken`*, or* `transfer_failed`

---

chandle `wrap_up_mode_auto on|off`

Description

Notifies the queue manager the type of `TRANSFER_METHOD` the caller is using to connect to an agent. Certain types of transfer methods can allow Amanda Portalto automatically detect when a hangup condition occurrs, thus allowing the queue manager to automatically go into wrapup mode for the agent.

---

chandle `transfer_ok`

Description

Notifies the queue manager the caller was successfully connected to the agent. Which in turn will change the status of the agent from `connecting` to `init_in_call`.

---

chandle `transfer_failed` [*reason*]

Description

Notifies the queue manager the caller failed to connect to the agent. Which in turn will change the status of the agent from `connecting` to `unavailable`. The call will return to the queue manager for further assignment. This function can also be called out of context, basically the normally context is when a caller is connecting to an agent. Out of context is any other time, in which the *reason* must be used. During these other times the caller can call this proc to update the status of its state. For example the caller can tell the queue manager that it dialed digits or is going to voicemail. This way the queue manager can update is statistics appropiately.

---

chandle `result` [`-nowait`]

Description

Returns the result of the implicit wait operation. When this call's disposition changes, then the result will become available. If `nowait` is specified then this will return immediately, otherwise this call will wait until a result is sent.

---

chandle `stop`

Description

Stops the waiting on the *chandle*.

Return values:
Empty string.

---

chandle `event`

Description

Returns the event handle to wait on the *chandle*.

Return values:
event.

---

chandle `hold_time` *seconds*

Description

This function indicates that the agent has put this call on hold for the indicated number of *seconds.* This information is recorded in the queue statistics database for the agent (not for the call).

---

chandle `agent_redirect`

Description

This function indicates that the agent is redirecting this call (transferring it). This information is recorded in the queue statistics database for the agent (not for the call).

Chapter 16

Connecting to External Databases (ODBC)

Amanda Portal needs the ability to talk to third-party databases to retrieve information. For example, suppose that technical support tracking information is stored in a third party database. Amanda Portal may need to query this database to find out information on a user's support call. This integration is done through the Open Database Connectivity or ODBC standard. This standard specifies an protocol that programs can talk to databases with. Databases and programs that understand this standard can be mixed and matched without recompiling or changing code.

ODBC functions are executed through the use of "handles" and member functions just like MMOs, VP devices, *etc.* When you delete the handle, your "connection" with the database goes away.

Normally, each operation is a separate transaction. This eliminates cumbersome `begin`, `commit`, `fetch` sequences. If you want to bundle up operations into one transaction, begin the transaction with the `begin` member function and end it with `commit` or `rollback`. You can then use the `fetch` operation to retrieve the results.

Because multiple ports and/or network clients may be executing SQL commands simultaneously, it is necessary that any ODBC drivers used be thread-safe. Unfortunately, not all of them are. The distribution of Amanda Portal includes Microsoft's installer for *Microsoft Data Access,* which includes ODBC version 3.51. All drivers included in that release are thread-safe. If you plan to use a driver other than one of these, you will need to verify that it is thread-safe before using it with Amanda Portal.

Here are the functions associated with ODBC:

`get_odbc_sources`

Description

Returns the different ODBC sources available in the system.

Return values:
An a-list where the keys are the database names found. The values of each key are the description of the driver used for each database.

Error codes:
`SQLERROR ...`

`connect_odbc` [`-user` *user*] [`-password` *password*] [`-timeout` *timeout*] [`-exclusive`] *data_source hvar*

Description

Connect to *data_source* using *user* and *password* (if needed) and return a handle in *hvar*. If `-exclusive` is given, then create a new connection to the database that will be broken when the variable is deleted; otherwise, share connections if possible. `-timeout` is only applicable without `-exclusive`. If given, then after all the variables using the connection are gone and *timeout* minutes occurs, the connection is released. The default is 5 minutes. If you are sharing the connection and you are not the first one to make the connection, `-timeout` is ignored. That is, only the first shared connection, which is the one which establishes the connection, gets to set the timeout value.

Return values:
Empty string.

Error codes:
`LOGINFAILED` *user password*
`DSRCNOTEXIST` *data_source*

Here are the member functions for the handle returned:

---

hfunc `tables`

Description

Returns a list of the tables in the database.

Return values:
An a-list where the key is the name of the table and the value is the type of the table.

---

hfunc `columns` *table_name*

Description

Returns information about each column in a table.

Return values:
An a-list of a-lists where the key is the name of the column and the value is an a-list with information about the column. The information returned can include `data_type`*,* `size`*,* `significant_digits`*,* `precision_radix`*,* `nullable`*, and* `description`*, depending on the database driver and the type of each column.*

---

hfunc `sql` [`-async`] *statement*

Description

Execute the SQL statement and return the results.

For `UPDATE`, `INSERT`, and `DELETE` statements, the value returned as `row_count` is either the number of rows affected by the request or `-1` if the number of affected rows is not available.

For other statements and functions, the driver *may* define this value. For example, some data sources may be able to return the number of rows returned by a SELECT statement or a catalog function before fetching the rows. Note, however, that many data sources cannot return the number of rows in a result set before fetching them; for maximum interoperability, applications should not rely on this behavior, but should instead ignore the `row_count` information and repeatedly call `fetch` until it returns `NODATA`.

Normally, the SQL command is executed synchronously, and control will not return to the Tcl interpreter until it has completed. The `-async` option causes the query to be run in a separate thread. In this case, you can use member functions `result` and `stop`, and use the *hfunc* with the `wait` command, just as with VP devices.

Return values:
An a-list with two elements, `row_count` *and* `col_count`*. You can use* `fetch` *to retrieve the results if the SQL command was a QUERY.*

Error codes:
`INVALIDSQL` *statement*

---

hfunc `procedures` [`-catalog` *catalog*] [`-proc` `proc`] [`-schema` *schema*]

Description

This function searches the database for the names of any stored procedures. By default, it returns all such procedures, even if they are procedures that the current database user is not permitted to execute. The results are returned as a result set, just as with an SQL query, so they can be enumerated using the `fetch next` command.

The arguments `-catalog`, `-schema`, and `-proc`, can be used to narrow the search. ODBC defines wildcard patterns, where `%` matches zero or more characters and _ matches a single character. The backslash can be used to make these two characters literal in the search when needed.

Return values:
An a-list with two elements, `row_count` *and* `col_count`*. You can use* `fetch` *to retrieve the results if the SQL command was a QUERY.*

---

hfunc `call` [`-result`] procedure_name [*-ptype param_name value_or_variable*]...

Description

The `call` command allows the user to invoke a stored procedure in the database. Stored procedures may return a value (as a function) or they may simply perform some operation, possibly returning values via output parameters (as a procedure). The `-result` argument, if given, specifies that the stored procedure will return a value; the returned value will be returned as the return value of the Tcl call. If `-result` is not specified, then the `call` command will return an empty string upon successful completion.

Stored procedures may have one or more parameters, and those parameters may be `-in`, `-out`, or `-inout`. Each parameter has a name. If an "in" parameter has a default value, and you wish to use that default value, then you may omit specifying that particular parameter altogether.

For each `-in` parameter, you must specify the parameter name and the *value* which is to be passed into that parameter.

For each `-out` parameter, you must specify a Tcl variable name into which the returned value should be stored.

For each `-inout` parameter, you must specify a Tcl variable name into which the returned value will be stored. This variable *must* already have a value which will be passed into the stored procedure when it is invoked.

Return values:
Whatever value is returned by the stored procedure, if it is a function, else an empty string in the case of true procedures.

---

hfunc `begin`

Description

Begin a new transaction. The transaction will not be finished until a `commit` or `rollback` is given later.

Return values:
Empty string.

---

hfunc `commit`

Description

Commit a transaction previously started with `begin`.

Return values:
Empty string.

---

hfunc `rollback`

Description

Rollback a transaction previously started with `begin`.

Return values:
Empty string.

---

hfunc `fetch` *dir*

Description

Fetch a row and return it. *dir* can be `next`, `prev` or an integer specifying the absolute row in the resultset. Some databases only support `next`.

Return values:
An a-list with the indices being the column names or the single string `NODATA` *when there is no more data.*

---

hfunc `fetch_info`

Description

Returns information about the result set that will be returned by the `fetch` command.

Return values:
An a-list of a-lists. The first a-list is ordered and the key is the name of the column. The value is an a-list with the following keys: `size`*,* `data_type`*,* `nullable` *and* `significant_digits`*.*

---

hfunc `get_isolation`

Description

Returns information about the transaction isolation levels available on this database connection in the form of an a-list. The keys of the a-list are `default`, whose value is the default isolation level, and `capabilities`, which is a list of possible levels. The levels are `read_uncommitted`, `read_committed`, `repeatable_read`, and `serializable`.

hfunc `set_isolation` *level*

Description

This function sets the transaction isolation level for this database connection to one of the four possible levels, `read_uncommitted`, `read_committed`, `repeatable_read`, and `serializable`.

Chapter 17

Miscellaneous

Here are some miscellaneous functions that don't seem to fit anywhere else.

`trace_out` [n]

Description

This function returns approximately the last n lines from the trace file, if the system is currently writing to a trace file. If there are fewer than n lines in the file, or if the file has "wrapped around," then `trace_out` may return fewer than n lines. If the file is empty or is not being written to, `trace_out` returns the empty string.

The logged-in mailbox must have the MONITOR privilege.

---

* `whoami`

Description

This function returns the mailbox number that the interpreter is currently logged in as. If not logged in, then the empty string is returned.

`shutdown` [`now`]

Description

This command is defined only for mailboxes with the SHUTDOWN privilege. If run without the `now` argument, a normal shutdown is begun, which takes up to `tmo_shutdown` seconds (a parameter from the Configuration Database, usually set to 60 seconds).

The command `shutdown now` will cause a very quick shutdown (equivalent to pressing "Exit Now" on the shutdown dialog; at most 5 seconds is allowed for the shutdown to complete).

Return values:
Empty string.

---

* `puts` [`-nonewline`] *string*

Description

This function is useful only in a *telnet* connection, usually for debugging purposes. It sends its argument *string* directly to the *telnet* client. Normally, the system will send a Return Linefeed pair after *string,* but this behavior can be suppressed by using the `-nonewline` option.

---

* `random` *limit*|`seed` [*seedval*]

Description

Generate a pseudorandom integer greater than or equal to zero and less than *limit*.

If `seed` is specified, then the function resets the random number generator to a starting point derived from the *seedval*. Often, the value of `[clock seconds]` is used for this purpose. By default, the seed is initialized to zero, so that consistent sequences of "random" numbers will be generated. Using such a constant seed value can be important in generating reproducible sequences for test purposes.

Return values:
Pseudo-random number in the desired range.

Error codes:
`NOTNONNEG` *limit*
`NOTINTEGER` *seedval*

---

* `sleep` *milliseconds*

Description

Sleep for *milliseconds,* or until a shutdown is initiated, and then return.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *milliseconds*

---

* `spawn` *script*

Description

Start a new thread and run the script in the new thread. The new thread runs as the person logged in if the person is logged in.

Return values:
Thread id of new thread.

---

* `enable_debugger` *boolean*

Description

Turns on or off the Tcl debugger within the current interpreter. It makes the debug window pop up. It must be called at the top of the call stack and `TclDebug.dll` must be loaded (use the `dlls` setting in the Configuration Database).

Return values:
Empty string.

Error codes:
`NOTBOOLEAN` *boolean*

`get_free_disk_space`

Description

Returns the (integer) percent free disk space across all MMO paths.

---

* `is_today` *time* [*days*]

Description

This function determines whether a *time,* expressed as Coordinated Universal Time, is yesterday, today, tomorrow, etc. If *days* is not specified, then it is assumed to be 0, and the function tests whether *time* falls on the same date as "today."

If `days` is specified, then it tests whether the given date, plus the specified number of days, would be today. Thus, to test if a date is "yesterday," one would use a value of `-1` for *days.*

---

* `queue_fax` [`-cover` *cover_text*] [`-notify` *computer_name*] *phone_num* [*mmo...*]

Description

This command allows you to queue a fax to be sent to a phone number. Each *mmo* must be either a fax MMO or an ASCII text MMO. Only logged in users can give the `-host` argument. If this argument is given, a popup response is sent back to the host when the fax is finished being sent. The popup response tells the user if the fax was sent successfully or unsuccessfully. If the fax send fails, it is tried `fax_max_retries` (from the Configuration Database; default value of 5) times before failing. The interval between retries is `fax_requeue_interval`.

This command uses the notify job scheduler. Therefore, if it is executed by a non-logged-in user, then the job is submitted to the queue to execute as the future_delivery mailbox, so this configuration database parameter must be defined, and this mailbox must exist, for anonymous users to be able to submit fax requests. The command is defined for anonymous users only on interpreters started to process inbound telephone calls. The job template used is called `FaxOut`. This template must exist for the job to be processed correctly. The MMOs are stored temporarily in the MMO Lookup Database so that they will persist until the fax has been sent; the `FaxOut` template is responsible for removing them.

Return values:
Empty string.

Error codes:
`CMDNOTEXIST` *mmo*
`CMDNOTHANDLE` *mmo*
`HWRONGTYPE` *mmo*
`MMONOTFORW` *mmo*

17.1 The `tokens` Command

The `tokens` command executes a subset of the programming tokens defined in the Amanda@Work.Group product. Amanda@Work.Group uses tokens as a short way of representing some behavior, such as hanging up the phone dialing some extension. Some of the tokens in Amanda@Work.Group cannot be implemented in Amanda Portal because they would violate security. For example, in Amanda Portal you can read from and write from the disk arbitrarily. Under Amanda@Work.Group this was acceptable using tokens because only the system adminstrator could set up tokens. Under Amanda Portal anybody can set up tokens and this would be a big security breach.

`tokens` *token_string...*

Description

Execute the token string.

Return values:
Returns 1 or 0 on success or failure. It may also raise an exception if it chains to another box.

The different tokens that are handled by Amanda Portal are as follows:

`-`	Pause for 1/2 second.
`@`	Stops processing of token string.
`+(`*variable*`[,`*value*`])`	Add or subtract a nubmer from the value stored in a variable. *variable* should be one of the port or global variables.
`=(`*variable*`,`*value*`[,`*start*`,`*end*`])`	Command that gives the variable the specified value. Use *start* and *end* to assign only part of a string to the variable.
`0`–`9`, `*`, `#`, `A`–`D`	Play the DTMF tone that corresponds to the specified digit.
`%A`	Expanded to the value of the `fax_dl_init` configuration setting.
`%B1`, `%B2`, `%B3`, `%B4`, `%B5`, `%B6`	Expanded to the serial number of the corresponding voice board.
`%C`	Expanded to the port number of the current caller.
`%D`	Expanded to the percentage of free disk space.
`%E`	Expanded to the contents of the current box's `EXTENSION` field (*i.e.*, the `EXTENSION` box setting).
`%F(`*field*`[,`*mailbox*`])`	Expand the name field (using the `User_Name` field in the MMO Lookup Table) or the comment field (using the `COMMENT` box setting) depending on the value of *field*. If *field* is `1` or `2`, expand the name field. (I know this is weird but it is the way Amanda@Work.Group works.) If *field* is `3`, expand the comment field.
`F`	Perform a hookflash.
`%G0`, `%G1`, `%G2`, `%G3`, `%G4`, `%G5`, `%G6`, `%G7`, `%G8`, `%G9`	The 10 global variables.
`G(`*mailbox*`)`	Causes a chain to the specified box. In Amanda Portal, it simply raises an exception with the error code `CHAIN` *mailbox*. The TUI then catches this code and takes the appropriate action.
`%H`	Expands to the value of the `CallerId` variable which contains the caller id.
`H`	Hang up.
`H(`*mailbox*`)`	Go to the specified mailbox if a hangup condition is detected.
`%I(`*field*`,`*msg_no*`)`	Expand the specified *field* from the specified message. *field* can be `D` for the date, `T` for the time and `F` for the "from" field.
`I(`*value*`,`*operator*`,`*value*`,`*mailbox*`)`	Control processing based on a condition. If the specified values and operator create a condition that is true, continue processing at the specified mailbox (*i.e.*, create a `CHAIN` exception). If the condition is false, the next token after the command is executed.

J(*box*,*phone_no*[,*tokens*])	Receive a fax file and sends it to the specified *box*. The *phone_no* and *tokens* arguments are ignored.
K<(*x*[,*value*])	Command that shifts the values of the %S variables to the left or right. *x* is a number from 0 to 20. When positive, shift left. When negative, shift right. *value* is the data to fill in the variables left empty by the shift.
KB(*frequency*,*millseconds*)	Beep. Play *frequency* for *milliseconds*.
KC(*mailbox*,*variable*)	Compare security code for mailbox with the contents of the variable. Returns 0 or 1 for a match.
KI(*target*,*source*,*variable*)	Search *source* string to see if it contains *target*. Fill in *variable* with 0 if not found in source; otherwise, fill *variable* with the position within *source* string at which *target* starts.
KR(*box*)	Record a message and send it to *box*.
L(*language*)	Set the language to *language*.
%M	Expands to the number of messages for the current box.
M(*greeting*,*repetition*,*delay*)	For *repeat* times, play *greeting*, wait up to *delay* tenths of a second for an input digit and then execute the MENU*n*_CODE.
%N	Expands to the number of new messages for the current mailbox.
O(*time*)	Go on-hook for the specified amount of time. Depending on the value used, you can cause a hookflash or hang-up. *time* is in tenths of a second.
%P	Expands to the previous mailbox.
P(*date*,D)	Say the specified number as a date. The number should be MMDDYY or MMDDYYYY (*e.g.*, 06261994) for June 26th, 1994.
P(*amount*,*currency*)	Say the amount as a currency. *currency* can be $ for dollars and cents, F for francs and centimes and P for pesos and centavos.
P(*number*,N)	Say the absolute value of a number.
P(*time*,T)	Say the specified number as a time of day. *time* should be in HHMM format.
P(A,*string*)	Say the characters of the specified string. For a space, say "space."
P(D)	Say the percentage of free disk space.
P(*greeting*[,*mailbox*])	Say the greeting for the specified mailbox.
P(M)	Say the number of messages for the current mailbox.
P(M*n*)	Play the message with the specified message number. Only audio messages can be played this way.
P(N[,*mailbox*])	Play the name and extension for the current mailbox or the specified mailbox.
P(*DTMF*)	Say a number as DTMF digits (*e.g.*, say 4–1–1 instead of four hundred eleven).
P(*prompt_no*,V)	Play a prompt in the current language.
P(R)	Say the DTMF digits entered by a caller requesting relay paging notification. These digits are stored in the relay_phone field of a message.
P(U[,*mailbox*])	Play name and extension recording of the specified or current mailbox.
P(V)	Say the digits in the variable field of a ctrigger record.
Q({*greeting*[#*mailbox*][,E]})	Ask the caller a series of questions and store all the caller's responses as a message in the current mailbox. Each answer will be a separate MMO. The E option gives the caller the opportunity to edit (review, rerecord, append or cancel) the previous group of answers.

%R	Expands to the DTMF digits entered by the caller requesting relay paging notification. It is the value of the relay_phone message field.
R(*greeting*[#*mailbox*], *variable*[, *timeout*])	Play the greeting from the current or specified mailbox and store the caller's DTMF entry as a number in the specified variable. The *timeout* is a number from 0 to 99 that represents the time in tenths of a second to wait for a DTMF entry after playing the greeting. The default is 1.2 seconds.
%S0 through %S19	Port specific variables where you can store and retrieve information.
S(*port*,[*string*],[*variable*], [*termination*],[*length*],[*timeout*])	Send and receive a string over a serial port. Terminate when the termination string is received, the maximum # of characters is received or a timeout occurs. C-type escape sequences are recognized. *variable* is the port specific or global variable to store the response. If not given, no response is stored. *termination* is a termination string. *length* is the maximum number of characters to receive. *timeout* is the maximum time in seconds that Amanda should wait for the first character and also between characters for the character to be received on the serial port.
%T	Expands to the number of seconds that the current call has been active.
%U	Expands to current mailbox number.
%V	Expands to the variable field of a ctrigger record.
%W	Expands to the day of the week as an integer. Number 1 is Sunday.
W(*n*[, *event*][, *mailbox*])	Wait for an event. If only the first argument is given, just wait that amount of time (in tenths of a second). The event argument can be P or V to wait for an answer (P used to be for pager and V was for voice) or T for dial tone. In the P or V case, *n* is the number of rings to wait for. In the T case, *n* is the number of seconds to wait for dial tone. If *mailbox* is given, chain to the given mailbox if the event does not occur.
%X	Expands to the codes needed to get the transfer dial tone. This is the setting of the dl_dtwait configuration parameter.
%Y	Expands to the current date in American format: MMDDYYYY.
%Z	Expands to the current time in 24-hour format: HHMM.

17.2 Time Functions

A number of functions exist for querying and manipulating time values:

---

* clock seconds

Description

Return the total number of seconds since 1970.

Return values:
The number of seconds.

---

* clock format *clock_value* [-format *string*] [-gmt *boolean*]

Description

Converts an integer time value returned by `clock seconds` or `clock scan` to human-readable form. If the `-format` option is present, the next argument is a string that describes how the date and time are to be formatted. Field descriptors consist of a % followed by a field descriptor value. All other characters are copied to the result. Valid field descriptors are:

%%	Insert a %.
%a	Abbreviated weekday name (Mon, Tue, *etc*).
%A	Full weekday name (Monday, Tuesday, *etc*).
%b	Abbreviated month name (Jan, Feb, *etc*).
%B	Full month name.
%c	Locale specific date and time.
%d	Day of month (01–31).
%H	Hour in 24-hour format (00–23).
%I	Hour in 12-hour format (00–12).
%j	Day of year (001–366).
%m	Month number (01–12).
%M	Minute (00–59).
%p	AM/PM indicator.
%S	Seconds (00–59).
%U	Week of year (01–52), Sunday is the first day of the week.
%w	Weekday number (Sunday=0).
%W	Week of year (01–52), Monday is first day of the week.
%x	Locale specific date format.
%X	Locale specific time format.
%y	Year without century (00–99).
%Y	Year with century (*e.g.*, 1990).
%Z	Time zone name.

If the `-format` argument is not specified, the format string "`%a %b %d %H:%M:%S %Z %Y`" is used. If the `-gmt` argument is present, the next argument must be a boolean value which if true, specifies that the time will be formatted as GMT. If false, then the local timezone will be used as defined by the operating environment.

Return values:
See above for return value.

Error codes:
`NOTNONNEG` *clock_value*
`NOTBOOLEAN` *boolean*

---

* `clock scan` *date_string* [`-base` *clock_val*] [`-gmt` *boolean*]

Description

Convert *date_string* to an integer clock value (see `clock seconds`). This command can parse and convert virtually any standard date/or time string, which can include standard time zone mnemonics. If only a time is specified, the current date is assumed. If the string does not contain a time zone mnemonic, the local time zone is assumed, unless the `-gmt` argument is true, in which case the clock value is calculated assuming that the specified time is relative to GMT.

If the `-base` flag is specified, the next argument should contain an integer clock value. Only the date represented by this integer is used, not the time. This is useful for determining the time on a specific day or doing other date-relative conversions.

The *date_string* consists of zero or more specifications of the following form:

time A time of day, which is of the form "*hh*[:*mm*[:*ss*]] [*meridian*] [*zone*]" or "*hhmm* [*meridian*] [*zone.*]" If no meridian is specified, *hh* is interpreted on a 24-hour clock.

date A specific month or day with optional year. The acceptable formats are "*mm/dd*[/*yy*]," "*monthname dd* [,*yy*]," "*dd monthname* [*yy*]" and "*day*, *dd monthname yy.*" The default year is the current year. If the year is less than 100, we treat the years 00-68 as 2000–2068 and the years 69–99 as 1969–1999.

relative time A specification relative to the current time. The format is *number unit*. Acceptable units are `year`, `fortnight`, `month`, `week`, `day`, `hour`, `minute` (or `min`), and `second` (or `sec`). The unit can be specified as a singular or plural, as in `3 weeks`. These modifiers may also be specified: `tomorrow`, `yesterday`, `today`, `now`, `last`, `this`, `next`, `ago`.

The actual date is calculated according to the following steps. First, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added. Next, relative specifications are used. If a date or day is specified, and no absolute or relative time is given, midnight is used. Finally, a correction is applied so that the correct hour of the day is produced after allowing for daylight savings time differences and the correct date is given when going from the end of a long month to a short month.

Return values:
Time in # of seconds since 1970.

Error codes:
`NOTNONNEG` *clock_value*
`NOTBOOLEAN` *boolean*
`INVALIDVALUE` *date_string*

---

* `uptime`

Description

Returns the number of seconds since Amanda Portal was brought up.

Return values:
The number of seconds.

---

* `is_yesterday` *cut*

Description

Given a Coordinated Universal Time value (the number of seconds since January 1st, 1970, GMT), return true (1) if that time occurred sometime during the previous day, and otherwise return false (0).

Error codes:
`NOTNONNEG` *cut*

---

* `is_today` *cut*

Description

Given a Coordinated Universal Time value (the number of seconds since January 1st, 1970, GMT), return true (1) if that time occurred sometime during the current day, and otherwise return false (0).

Error codes:
`NOTNONNEG` *cut*

17.3 Terminating Threads

Threads are divided up into categories and there are functions for inspecting and terminating threads in the system. They are as follows:

`get_thread_info` [`-me`] [`-category` *category*]

Description

Return information on each thread running in the system. If *category* is given, only return information on the threads in each category. *Category* can be `monitor`, `notify`, `template`, `AutoScheduler`, `spawn`, `phone`, `network`, `sound_card`, `startup`, and `pop`. If `-me` is specified, then only information about the current thread is returned.

Return values:
An a-list of a-lists is returned. The first a-list has an index for each thread. The index is the thread id. The value at each index is another a-list with the indices `box`, `category` *and* `host`. `host` *is only given if the user is connected through the network.*

Error codes:
`INVALIDCATEGORY` *category*

`kill_user_thread` [*box*]...

Description

Kills all threads for the box(es) given. You need the KILL privilege to execute this call and you must be an (improper) ancestor of the box.

Return values:
Empty string.

Error codes:
`NOTNONNEG` *box*
`BOXNOTEXIST` *box*
`PERMDENIED`

`kill_thread` [*thread_id*]...

Description

Kills the threads with the given *thread_id*s. You need the KILL privilege to execute this command and the threads need to be running as descendents of your box.

Return values:
Empty string.

Error codes:
`THREADNOTEXIST` *thread_id*
`PERMDENIED`

17.4 Logging

There are a couple of functions for logging information to files or the console. They are as follows:

---

* `wlog` [*arg*]...

Description

Write a log message to the Amanda log file.

---

* `wtrace` [*arg*]...

Description

Write a trace message to the Amanda trace file.

17.5 Speech Processing

The following command is loaded only if you load the Speech Recognition DLL (AmaSapi5).

`compile_grammar` *mmo grammar*

Description

Compiles *grammar*. If an error occurs, return an error indication in the usual way. Otherwise, store the compiled grammar into the specified *mmo* for future use in Speech Recognition. Grammars are used in speech recognition with the VP `recognize` command.

Return values:
Empty string.

Error codes:
`PERMDENIED`
`GRAMMARCOMPILEERR` *grammar*

The following member functions for voice processing resources become enabled:

---

* `create_sr` *engine_var*

Description

The `create_sr` function can be used to create a speech recognition engine which can be used with the `recognize` function, described below, to perform speech recognition functions. The engine is released when the *engine_var* becomes unset.

---

* `recognize` *record_flags* [`-beep` *boolean*] [`-sr_stream` *engine*] [`-threshold_low` *value*] [`-threshold_high` *value*] [`-start` *rules*] *grammar_mmo*

Description

Recognize speech against the grammar. The grammar identified by *grammar_mmo* must have been previously compiled with `compile_grammar`. The return value will be the same as for the `record` member function if the operation is interrupted in some way (timed out, stopped, caller pressed a digit, etc.). Otherwise, it returns an a-list of information detailing what phrase was recognized (by command number), the confidence level of the recognition (from -100 to +100), and any strings embedded in the grammar itself (for which care should be taken that they will appear to be an a-list).

The `-beep` argument is a convenient equivalent of the standard `-nobeep` flag to the `record` function. It takes any Tcl boolean value as its argument.

The `-sr_stream` argument can be used to tell the `recognize` function to use a pre-existing speech recognition engine, specified as *engine*. If this argument is not used, then an engine is automatically allocated and de-allocated by the `recognize` function. The use of a pre-existing engine can decrease the time between when the `recognize` function is called and when it begins recognition since an engine does not have to be allocated (and potentially, created, which can be time-consuming). Engines can be allocated using the `create_sr` function described above.

The `-start` argument can be used to specify which grammar rule the recognition is applied to. If this argument is not specified then all active rules are applied.

The `-threshold_low` and `threshold_high` are used to control the success rate of the recognition. The values can be between 0 and 100. The default is 50 and 75 respectively. The actual value recieve on a particular recognition is returned in the `confidence` value.

---

* `speak` *play_options* [`-gender` *male/female/neutral*] [`-speaker` *name*] [`-language` *langcode*] [`-sublanguage` *sublangcode*] phrase...

Description

Perform text to speech on the phrase(s) specified. Most play options are supported: `-nodtmf`, `-clear`, `-volume`, `-speed`, `-term`, `-noretain`, `-bargein`, and `-maxpause`.

The `-gender` option specifies a desire for the type of voice to be employed. The default is a neutral voice (meaning that you don't care), but `male` or `female` can be specified. There is no guarantee that the requested gender will be available, as this depends on the available TTS engine(s) which are installed on the system.

The `-speaker` option allows you to specify a specific engine by speaker "Name". The manufacturers of different text to speech engines give each variation of their engine(s) a name, such as `Paul` or `Mary`. If you specify the speaker name, it must match exactly with the engine's name (case sensitive).

The `-language` option allows you to specify a specific engine by the language it supports. The default value for this parameter is 9, defined by Microsoft as the code for English. This is the only language code supported by the Microsoft SAPI TTS Engines.

The `-sublanguage` option allows you to specify a dialect of the primary language selected with `-language`. The default is 1, US English. Microsoft defines many other values, specific to each language. For instance, UK English is 2, Australian English is 3, Canadian English is 4, New Zealand English is 5, and Irish English is 6. Similar sublanguage constants are defined for all other language codes which are supported by the `-language` argument. The SAPI TTS engines provided by Microsoft support only US English.

The return values and errors are the same as for the `play` member function.

You can also set the configuration parameter `sr_wait`. This parameter in milliseconds will wait this long for buffers to be read in. If no buffers have been read in in this amount of time then this function will stop recognizing and return a result.

The *engine_var* returned by `create_sr` has the following two member functions:

---

* *sr* `handle`

Description

This is an internal command used by the voice processors to get a direct handle on the object. The return value is the address of the object. This function should not be called.

---

* *sr* `grammar` *rule property var_list*

Description

This allows modifications to dynamic rules of the grammar. *rule* is the dynamic rule to modify. *property* is the property name (PROPNAME attribute) of the elements to add. *var_list* is a list of elements to add. For example this can be used to add the names of the mailboxes into the grammar at run time.

---

* *sr* `set_profile` *profile attrib*

Description

This set the current profile for the recognition engine to use. In general a profile represents a specific user of the system. So, when a user logs in then the appropiate profile can be retrieved for that user for optimum performance. *profile* is the name of the profile to make current. *attrib* is an a-list of attribute values that define this profile for this particular type of recognition engine. For example, attributes can be `GENDER` (with values of `Neutral`, `Male`, or `Female`); `AGE` (with values of `Child`, `Teen`, `Adult`); or `ADAPTATION` (with values of `0` or `1`).

17.6 Web Client Access

The optional `WebClient` module allows access to Web servers at the Tcl level. This can be used to retrieve information and/or to post information ("Submit" it) to a Web site. The first command is merely a helper command which can be used to set up a query for a POST operation, while the second one can be used to perform a GET, POST, or HEAD operation. These commands are based on the similar commands which are in the `http` package included with Tcl 8.*x*.

`http_formatquery` *key value...*

Description

This function formats a series of one or more key/value pairs into the format required for POSTing a query to a web site. The resulting string can be used with the `-query` argument to the `http_geturl` function.

`http_geturl` *url* [`-headers` *key value list*] [`-query` *q*] [`-timeout` *ms*] [`-type` *mimetype*] [`-validate` *boolean*] [`-secure`] [`-user` *username*] [`-password` *pass*] [`-useragent` *string*] [`-proxyhost` *host*] [`-proxyport` *port*] [`-accept` *mimetypes*]

Description

The `http_geturl` function allows access to a Web server. Its options are:

The `-headers` option is used to add extra headers to the HTTP request. The *key value* list argument must be a list with an even number of elements that alternate between keys and values. The keys become header field names. Newlines are stripped from the values so the header cannot be corrupted. For example, if keyvaluelist is Pragma no-cache then the following header is included in the HTTP request: Pragma: no-cache

The `-query` option causes `http_geturl` to do a POST of the query *q*.

The `-type` option allows you to specify the mime-type which will be used during a POST operation. The default is `application/x-www-form-urlencoded`.

The `-validate` option allows you to specify that a HEAD request should be posted.

The `-useragent` argument allows you to specify the value value of the User-Agent header in the HTTP request. The default is `Amanda`.

The `-proxyhost` argument allows you to specify a proxy host through which the query should be posted. The default is to use the address given in the specified *url*.

The `-proxyport` argument allows you to specify a non-default port number to be used, either for proxy or non-proxy queries. The default is to use port 80.

The `-secure` option causes the connection to the web server to be made to port 443 instead of port 80, and the data passed between the client and server is encrypted via SSL.

The `-user` and `-password` options can be used when the server requires authentication of the client.

If neither `-validate true` nor `-query` are specified, then an HTTP GET operation is performed.

`http_parse_xml` *xml*

Description

This function parses an XML string into the equivalent Tcl a-list format, which is returned as its value.

17.7 TCP Client Connections

Amanda provides commands such as `http_geturl` or POP commands for accessing specific types of servers. Sometimes, however, it is necessary for Amanda to connect to another computer in a specialized way. The TCP Client Connection feature allows you to do this.

`create_tcp` *server port handle*

Description

The `create_tcp` function allows you to create a TCP connection. To be able to call it, you must be logged in or the current interpreter must be a "channel interpreter" which is running in response to an incoming telephone call. If `create_tcp` fails, it returns the failure reason and TCL_OK. On success, it returns an empty string (no error message).

Once you have successfully connected to a server, the *handle* variable can be used to send and/or receive on the socket. Its subcommands are:

---

* *handle* `send` string

Description

This sends *string* to the server at the far end.

---

* *handle* `receive` [`--async`] [`--max` *max*] [`--immediate`] [`--timeout` *tmo*]

Description

This command causes the system to do a |recv()— operation on the handle. There are several options to the receive function.

If the receive is performed asynchronously, then the command will return immediately. Subsequently, you may use the `stop` and `result` functions to access the results of the receive.

The `-max` option sets the maximum number of characters to wait for. Once that many have been received, then the receive function ends. The `-immediate` option, which may be combined with `-max` and/or `-timeout`, says that once one packet of data has been received, then the receive operation should return instead of waiting any further. The `-timeout` option specifies the maximum number of milliseconds that the system will wait without receiving anything on the socket, before returning whatever has been received to that point.

---

* *handle* `stop`

Description

This function causes a previously-issued asynchronous receive command to be stopped, if it is still running.

---

* *handle* `result`

Description

This function returns the results of a previous-issued asynchronous receive function. If the function is still running, then it will block until the function finishes, and then return the results.

17.8 VoIP Appliance Access

The following commands are used to communicate with the VoIP Appliance (Indavo) are within the ILinkMan.dll. The commands implemented by this module are Indavo-specific, but they exemplify the type of support modules which can be easily added to the system to support any VoIP edge device.

The Indavo is a special PBX that uses the Internet as the voice communication path. The Indavo can be configured through the following commands:

---

* `connect_indavo` *ip handle_var*

Description

Open a connection to an indavo box. The *ip* is the TCP/IP address of the Indavo box. The *handle_var* is the variable that represents the connection object and has the commands that follow this command.

Return values:
Empty string.

Error codes:
`ALREADY_CONNECTED`
`FAIL_LOAD_PARAM`
`ILINK_FAIL_CONNECT`
`ILINK_VERSION_OLD`
`ILINK_CORRUPT`

---

* *handle_var* `save`

Description

Saves all the settings to the Indavo box. This might return an `INDAVO_RESET` error. This means that the settings were saved, but the indavo is being reset and in doing so the connection is lost. If no reset was done then an empty string is returned. Note that the indavo box sends a reset recommendation when certain fields are set, mainly the ones in *set_config* and *set_ring_group.* If the Indavo box sends that recommendation, then it is reset and the socket is close.

Return values:
Empty string.

Error codes:
`FAIL_LOAD_PARAM`
`FAIL_SAVE_PARAM`
`ILINK_FAIL_CONNECT`
`ILINK_CORRUPT`
`ILINK_RESET`

---

* *handle_var* `set_ring_group` *group* [*field value*]...

Description

This sets the *field* of a ring group. A ring group is a group of extension ports. The *group* is 1 based, ranging from 1 to 6. The `ext_list` *field* must be a tcl list of 12 boolean elements. Must call `save` after all the set_*xxxx* have been called. *fields* can be any of the following: `phone`, `rna`, `busy`, and `ext_list`.

Return values:
Empty string.

Error codes:
ILINK_INVALID_GROUP
ILINK_GROUP_RANGE
FAIL_GET_PARAM

---

* *handle_var* `get_ring_group` *group* [*field*]...

Description

This gets the *field* of a ring group. A ring group is a group of extension ports. The *group* is 1 based, ranging from 1 to 6. If no fields are specified then all fields are returned. *fields* can be any of the following: `phone`, `rna`, `busy`, and `ext_list`.

Return values:
An a-list of the fields and their values.

Error codes:
ILINK_INVALID_GROUP
ILINK_GROUP_RANGE
FAIL_GET_PARAM

---

* *handle_var* `set_port` *port* [*field value*]...

Description

This sets the *field* of an extension port. The *port* is 1 based, ranging from 1 to 12. Must call `save` after all the set_*xxxx* have been called. *fields* can be any of the following: `phone`, `name`, `rings`, `rna`, `busy`, `fwd_phone`, and `call_waiting`.

Return values:
Empty string.

Error codes:
ILINK_INVALID_PORT
ILINK_PORT_RANGE
FAIL_GET_PARAM

---

* *handle_var* `get_port` *port* [*field*]...

Description

This gets the *field* of an extension port. The *port* is 1 based, ranging from 1 to 12. If no fields are specified then all fields are returned. *fields* can be any of the following: `phone`, `name`, `rings`, `rna`, `busy`, `fwd_phone`, and `call_waiting`.

Return values:
An a-list of the fields and their values.

Error codes:
ILINK_INVALID_PORT
ILINK_PORT_RANGE
FAIL_GET_PARAM

---

* *handle_var* `delete_port` *port*

Description

This clears out an extension port. The *port* is 1 based, ranging from 1 to 12. This is just a special form of `set_port` that sets everything to 0 or empty. Must call `save` after all the set_*xxxx* have been called.

Return values:
Empty string.

Error codes:
ILINK_INVALID_PORT
ILINK_PORT_RANGE
FAIL_GET_PARAM

---

* *handle_var* `set_config` [*field value*]...

Description

This sets the basic configuration settings of the indavo box. Must call `save` after all the set_*xxxx* have been called. *fields* can be any of the following: `911_redirect`, `411_redirect`, `voicemail_redirect`, `base_ext`, `base_group`, `operator_ext`, `jitter`, and `comfort_noise`.

Return values:
Empty string.

Error codes:
FAIL_SET_PARAM

---

* *handle_var* `get_config` [*field*]...

Description

This gets the basic configuration settings of the indavo box. If no fields are specified then all fields are returned. *fields* can be any of the following: `911_redirect`, `411_redirect`, `voicemail_redirect`, `base_ext`, `base_group`, `operator_ext`, `jitter`, and `comfort_noise`.

Return values:
An a-list of the fields and their values.

Error codes:
`FAIL_GET_PARAM`

17.9 COM/OLE

Some modules require COM to be activated. Currently AmaSAPI5 is the only such module. In the future there may be others. COM needs to be initialized and uninitialized in a consistent manor within a particular connection instance to Amanda (thread). The following function within the core of Amanda will control that.

---

* `enable_com` *multithreaded*

Description

This function will enable COM for this particular connection (thread). Consult the particular module's documentation on what the value of *multthreaded* should be. This boolean value specifies if COM should be initialized to be used with several threads, or only this thread (apartment). For example, AmaSapi5, it can be 0 for apartment. This function can not be called mutliple times specifing a different *multithreaded* value.

Error codes:
`CONNREFUSED`

Appendix A

Error Codes and Messages

When an exception occurs, the error code and error message are initimately tied together. With the exception of `USAGE` and `BOARDERROR` error codes, the error message will always be the same. The parameters in each error code may vary though, even for the same error code. You need to look at the documentation for the API function to see the exact value returned.

Here are the error codes and associated error messages:

AGENTATTACHED	Cannot delete queue when agent is attached.
AGENTNOTAVAIL	Agent not available.
AGNOTENABLED	Agent not enabled on queue.
BADLISTLEN	Bad list length.
BADOPTION	Bad option.
BOARDERROR	A board error occurred.
BOARDNOTEXIST	Board does not exist.
BOXEXISTS	Box exists.
BOXHASCHILD	Cannot delete a box with children.
BOXHASMSGS	The destination box has messages.
BOXLOGGEDIN	Cannot delete a box that is logged in.
BOXMANDATORY	You must give the box argument when not logged in.
BOXNOTEXIST	Box does not exist.
BREQNLOGIN	You must be logged in or specify -box.
CALLNOTOUTSTAND	There is no outstanding call to accept.
CMDNOTEXIST	Tcl command does not exist.
CMDNOTHANDLE	Tcl command is not a handle.
CONFNOTEXIST	Configuration value doesn't exist.
CONNBROKEN	Connection broken.
CONNREFUSED	Connection refused.
CONVFAILED	Conversion failed.
CTIMENOTSET	Call time not set.
DSRCNOTEXIST	Data source does not exist.
FAX_ANSWER	A fax answer tone was heard during PCPM.
FAX_INITIATION	A fax initiation tone was heard.
FIELDNOTEXIST	Field does not exist.
FILENOTFOUND	File not found.

FILEOPENFAIL	File open failure.
FORWNOTSTORE	Cannot store non-forwardable MMOs.
FUNCNAMETOOLONG	Function name too long.
GBOXLIMIT	Maximum number of child mailboxes reached.
GRAMMARCOMPILEERR	Grammar compile error.
GRAMMARNOTEXIST	Grammar does not exist.
HANGUP	A hangup was detected.
HOSTNOTEXIST	Host does not exist.
HREADONLY	Handle is read-only.
HWADDRFETCH	Cannot fetch hardware address of machine.
HWRONGTYPE	Handle is the wrong type.
IDNOTEXIST	Id does not exist.
INDEXMISSING	The index is missing.
INVALIDCATEGORY	Invalid category.
INVALIDCONFVALUE	Invalid configuration value.
INVALIDCONFVALUE	Invalid folder number.
INVALIDHOSTNAME	Invalid host name.
INVALIDKEY	Invalid key.
INVALIDPASSWORD	Invalid password.
INVALIDRESTYPE	Invalid resource type.
INVALIDSQL	Invalid SQL statement.
INVALIDTIME	Invalid time.
INVALIDTYPE	Invalid type.
INVALIDVALUE	Invalid value.
INVRECIPIENT	Invalid recipient.
KEYMANDATORY	Key is mandatory.
KEYNOTEXIST	Key does not exist.
KILLED	Interpreter was killed.
LANGNOTEXIST	Language does not exist.
LOGINFAILED	Login failed.
MALFORMEDFILE	Malformed file.
MAPEXISTS	Map exists.
MAPNOTEXIST	Map does not exist.
MAXHANDLES	The maximum number of handles has been reached.
MAXLANGUAGES	Maximum # of languages exceeded.
MAXMMOS	Maximum MMO count exceeded.
MEMALLOC	Memory allocation failure.
MISSINGPARAMETER	Missing parameter.
MMODISKFULL	Disk space for MMOs is filled up.
MMONOTFORW	MMO not forwardable.
MMONOTWRITABLE	MMO not writable.
MSGNOTEXIST	Message does not exist.
MSGNOTFORW	Message is not forwardable.
MULTQATTACH	Cannot attach to a queue multiple times.
NEVERASYNCCMD	You never ran an asynchronous command.
NOCURMSG	No current message.
NOLANGSEL	No language selected.
NOMEANING	The operation has no meaning.

NONAUDIOMMO	Not an audio MMO.
NONFAXMMO	Not a fax MMO.
NONTEXTMMO	Not a textual MMO.
NOOPINPROGRESS	No operation in progress.
NOQUERY	No active query.
NOTHREAD	No thread running.
NOTALIST	Not an a-list.
NOTATTOPFRAME	Not at the top Tcl frame.
NOTBOOLEAN	Not a boolean.
NOTINTEGER	Not an integer.
NOTLIST	Not a list.
NOTMMO	Not an MMO.
NOTNONNEG	Not a non-negative integer.
NOTNONNEG	Not a positive integer.
NOTRANSFERSET	Cannot transfer a call marked as non-transferrable.
OFFHOOK	Port is off hook.
OPINPROGRESS	Operation already in progress.
OUTOFRANGE	Out of range.
PERMDENIED	Permission denied.
PORTCONFLICT	Cannot give two variables the same port number.
PORTNOTEXIST	Port does not exist.
PORTPRIVILEGED	Cannot open privileged port.
PRIVNOTEXIST	Privilege does not exist.
PROCINUSE	Procedure is in use.
PROCNAMETOOLONG	Procedure name is too long.
PROCNOTEXIST	Procedure does not exist.
PROMPTNOTEXIST	Prompt does not exist.
QEXISTS	Queue already exists.
QUEUENOTEXIST	Invalid queue.
QUEUEFULL	The queue is full.
RESALLOC2BIG	Not enough resources of the type given in the system.
SYSBOXFULL	Maximum number of system boxes reached.
THREADNOTEXIST	Thread does not exist.
TONE	A tone was heard.
UNKNOWNERROR	Unknown error.
UNKNOWNTYPE	Type is unknown.
UNSUPPORTED	Operation is unsupported.
USAGE	Bad usage.
VALUENOTEXIST	Value does not exist.
WRITEERROR	Write error.
WRONGAUDIOFORMAT	Wrong audio format.

Index