GNU Bayonne Script Programming Guide II

David Sugar
GNU Telephony.
sugar@gnu.org, http://www.gnutelephony.org

2003-12-20

Contents

1 Introduction

GNU Bayonne 2 is a script driven telephony application server that comes with it's own built in scripting language. Many people ask why GNU Bayonne has it's own scripting language rather than using an existing one. There are several reasons for this.

First, Bayonne scripting is deterministic. In some scripting languages, expressions can be parsed and evaluated as arguments and mid-statement, including expressions that are long and have no fixed runtime duration. GNU Bayonne 2 requires realtime response and non-blocking behavior; it cannot directly execute things that have undetermined execution times.

To reduce system load and support a very high number of concurrent script sessions (up to 1000), I looked at being able to execute script statements on the leading edge of a callback event rather than requiring a seperate thread to support each interpreter instance. The possibility of requiring up to 1000 seperate additional threads to support scripting on large port capacity systems seemed prohibitive. Most script interpreters, besides being non-deterministic in their behavior, also do not support direct single stepping in that manner.

In Bayonne scripting, it is also possible to effectivily isolate and seperate execution of statements that require extended duration to execute or that might block (such as I/O statements) from those that can be executed quickly from a callback handler in realtime. A thread can then be used to support execution of a statement that may otherwise block or delay realtime response. Most scripting languages do not directly support seperation of execution for "quick" and "slow" statements since they are not directly concerned with the effect of blocking operations.

A number of other special optimizations also exist in Bayonne scripting to reduce heap fragmentation. This was deemed important when supporting a very large number of running instances. Heap fragmentation is resolved both by using fixed size and unmovable symbol entries, and by allocating memory for running instances in pages which are then doled out to individual symbols on demand, rather than performing individual new/malloc operations for each symbol.

Finally, I wanted to execute scripts immediately and from memory context. The reason for this is that loading a script file from disk is of course a potentially blocking operation, and is an operation that would need to be frequently done in most scripting systems. Bayonne scripting forms a single interpreter image by compiling all scripts at once directly into memory. Furthermore, since I did not wish to have the server "down" to load a new script image, I can maintain more than one such core image; when new scripts are loaded, any calls currently in progress continue to use the previously active script image from memory. New calls are offered the script a new image if the server has been asked to load one. When the last of the older calls complete, then the older image is purged from memory. This permits continual server uptime even while replacing scripts that are held in memory.

2 Script and Module Files

There are two types of script files used in Bayonne scripting. A normal script is written as a ".scr" (Bayonne Script) file. All ".scr" files found in the Bayonne directory are compiled as the server is started. The server can also be used to compile and execute a single script file only, or a group of script files which are passed directly as command line arguments.

The second type of Bayonne script file is known as a "Bayonne macro" or .mac file. Like modules in perl, .mac files are compiled from a common central script directory when requested and used in Bayonne scripts. This happens when using the "requires" or "import" statement in your script. Macro files often are used to contain function libraries which may then be invoked from your script once imported. The functions are considered part of the named module, and hence, if one imports "mymodule.mac", one can reference it's contained functions and non-private labels through "mymodule::label".

3 Statements and syntax

Each script statement is a single input line in a script file. A script statement is composed of four parts; an event flag as appropriate, a script command state-

ment, script command arguments, and keyword value pairs. White spaces are used to seperate each part part of the command statement, and white spaces are also used to seperate each command argument. Each script file is considered a self contained application, and may itself be broken down into named sections that are labelled and individually referenced.

Line input is not limited to 80 characters. However, to make it easy to edit very long lines, they can be split up by using a at the end of a line to join multiple lines together.

Script commands may be modified by a special .member to describe or activate a specific subfunction. For example, a "foo.send" command, if it existed, would be different from but still related to a plain "foo" command. Members are often used for special properties, to specify an offset value, or the size of script symbols that are being created.

Command arguments are composed either of literal strings or of references to symbols. Symbols are normally referenced by starting with the "%" character and can be typed in a variety of ways explained below. A group of arguments may also appear within double quotes, and these will be expanded into multiple arguments that are composed of literal string constants and the substituted values of variables that are referenced. Double quoted arguments are most often used to compose strings that are to be parsed by other subsystems (such as sql statements) or to format output for logging.

Since sometimes it is useful to refer to arguments that may appear as strings or other forms of content, 's be used to enclose a literal section. A literal section is passed as a single string argument regardless of the content it contains. Hence, while "my value is %1" would form two string arguments, 'my value is ', and the value of %1, the literal expression my value is %1 would return a single string argument in the form 'my value is %1' unmodified. 's may also be used to pass character codes that are normally not supported as command arguments.

A complete script defines a labeled section of a script file, such as a script section, procedure, or function, the script statements the labelled section contains, and the statements found in any event handlers associated with the label. The script file itself has a default label under the name of the script, and any statements that appear before labels are used are associated with a label named by the script

file itself. When event handlers are attached to labeled sections of a script, and control passes to the event handler, the script blocks all remaining events until a new script section is entered.

4 Labels

Labels are used to segment script files into individual script entities. These entities are all stored and referenced from a common hash table. Script labels that are the default script code for a given .scr file are stored under the name of the script itself. Within the script, local labels may be used. These local labels are preceded by a keyword qualifying their scope.

The scopes used for labels include "private, protected, and public". A label described as "private" may only be referenced from within the current source file. A label tagged as "protected" may be called or referenced by other scripts, but may not receive initial control when an application is started by Bayonne. Only a label tagged as "public" (or "program") or "service" may receive initial control.

A script file that has commands which need to branch to a private label or any label local in the current source file can do so by specifying the name of the label, either with or without preceeding the label with ::. Hence, if there is a "private test" entry in "your.scr", another script command can branch to the test: entry directly by using either ::test or test, as in "goto ::test" or "goto test", for example.

Labels within other script files, unless made private, may also be referenced and branched to directly. If myscr.scr has a script section "protected test", I can branch to it by refering to myscr::test. Hence, I could use "goto myscr::test" for example.

Script labels may also be exported into another script. This can be useful when making a script label private but wishing to make it available as or part of another script file. This is done by using the full name of the label as part of the private, protected, or public keyword, including the script file, ::, and the local target name. Hence, to export a private section of code to be named "mycode" into "hisscript.scr" directive, we can use "private hisscript::mycode".

5 Functions

A function is a named or labeled section of script that can be invoked from other scripts such that when the function completes, control is returned to the original calling script where it left off. Functions are defined as a label with the function keyword, as in "function myfunction" for example.

A function supports the use of inhereted event handlers. Hence, if the script file that calls a function has for example, a keyboard trap handler for the "1" key, that trap handler will still be active when the function is being executed. If the function call occured from within the "1̂" handler itself, then since dtmf is normally masked in scripts that are in handlers, the inhereted "1" handler will also be masked in the function call. Function calls can still define their own local event handlers, which are then used, and can define new local event handlers even if the script that called them had those handlers disabled. Hence, inhereted event handlers are only used if an event occurs which the local function does not define a handler for, but the calling script would normally execute.

Another special property of functions is their operation with the "goto" statement. When a function transfers control to a label with "goto", the stack and all local variables for the function are automatically cleared. A function that goes to a "label" cannot return control to it's calling script. When a function goes to another function through a "goto" statement, local variables are kept intact, and when that function returns, control resumes from the calling script, unless, of course, it has a "goto" statement to a label.

A function is invoked through a "call" statement, and can receive command arguments. These arguments may be passed either by reference or by value. Command arguments also may also be both positional and associative. Associative command arguments are passed with keyword= values, which may then be filled into locally scoped variables. Initial default associative command arguments may be defined with the function label, as in for example "function myfunc retries=3", where "retries" is given a default value of 3, unless a different value is passed by the function call.

Functions may be invoked either through a call statement, or directly. To call a function in a foreign module, one can use either “call themodule::myfunct retries=7”, for example, or directly state the function name as if it were a command, as in “themodule::myfunct retries=7”. Functions that are local to the current source file may also be invoked as a keyword, using “*::” for the module name, as in “*::myfunc retries=5” for example. See “call” and “return” for further information on functions.

6 Symbols

Bayonne scripting recognizes four kinds of symbols; constant ”values” (literals), %variables, #evaluated and @indirection. In addition, Bayonne recognizes compile time substitutions, known as $names, which can substitute to any of the above four. A literal can be ”string” literals, which are double quoted, numeric literals, such as 123, which are without quotes, and {as-is string} literals encased in {}’s. (see Sec.2).

A %variable can be defined either as having content that is alterable or that is constant. Some %variables are automatically initialized for each and every telephone call and some are created automatically in local scope when a function call is made, while others may be freely created by scripts themselves either in global scope, or locally within a function. However, all variables and values stored, with the exception of special shared variables, are automatically cleared at the end of each telephone call.

Constant variables are declared with the “const” keyword. Sometimes this is useful to have a non-modifiable variable when defining a value that may have an overiding default which can be asserted at an earlier point in the code. This is especially important in subroutine calls and argument passing, as will be explained later. However, most often, variables are used to store changable values.

The # operator is a special case. When used, the numeric value of the symbol is returned. The meaning of this depends on the symbol type. Generic symbols return their current string length when #symname is used. Others may return numeric values based on their type. For example, a “position” variable will return the number of seconds in a timestamp position, and a “date” variable will return

a julian number.

Bayonne also recognizes symbol scope. Local symbols are created within symbol scope, resulting in unique instances within functions or subroutines. In addition, all local symbols in symbol scope are removed when returning from a subroutine. Globally scroped symbols are visible everywhere under the same instance. A "global" symbol uses "." notion, as in %myapp.name, where a "local" symbol uses a simple name, as in %name, for example. Generally it is suggested that a given script should organize it's global symbols under a scriptname.xxx format to make it easier to read and understand. This can be done by using the special '.' notation. Hence, within the file "myscr.scr", ".myvar" can be used in place of or as a shortcut for %app.myscr.myvar. This also prevents collisions with macro files, which expand a '.' name into %mac.mymac.myvar.

You can concatenate symbols and constant strings with either non-quoted whitespace or the comma operator. For example,

```
set %a ``a'' ``b'', ``c''
```

results in %a being set to "abc".

Finally, there is a shortcut notation to create global symbols that are associated with a single script file. If we have a script named foo.scr, and wish to create a bunch of global symbols related to "foo", we do not have to create %foo.xxx, %foo.yyy, etc. Instead, we can refer to these script global names simply as .xxx and .yyy. These will be expanded at compile time back to %app.foo.xxx and %app.foo.yyy. When in another script file, one can reference the full name, %app.foo.yyy directly, while the shortcut form can be used within the script file, or within things exported as foo.

The following variables are commonly defined:

%script.error	last error message
%script.token	current token seperator character
%script.home	same as %session.home

%session.id	global call identifier
%session.pid	call identifer of parent session
%session.callid	call instance identifier
%session.timeslot	server timeslot instance
%session.deviceid	name of timeslot endpoint
%session.voice	current voice in effect
%session.position	timestamp from last audio replay/record operation
%session.driverid	name of session device driver
%session.spanid	span number of device
%session.tid	transaction id
%session.rings	number of rings before pickup occured
%session.date	date script was started
%session.time	time script was started
%session.duration	duration of call so far
%session.type	call session type
%session.interface	type of interface session is running on
%session.bridge	type of briding supported on this driver
%session.digits	currently collected dtmf digits in the digit buffer
%session.count	count of number of digits collected
%session.caller	generic identity of calling party
%session.display	display name of caller
%session.dialed	number dialed
%session.info	call info, such as ii digits, or sip peering
%session.callref	common billing id for networked calls
%server.timeslot	total timeslots on bayonne server
%server.version	version of bayonne server
%server.software	software identifier; “bayonne2”
%server.platform	platform running under
%server.driver	which driver we are running
%server.node	node id of our server
%sever.location	location set for tones
%server.voice	default voice library
%sql.driver	name of loaded sql driver
%sql.current	current row cursor position
%sql.changes	number of rows effected by last command
%sql.rows	number of rows returned in last query
%sql.cols	number of cols returned in last query
%sql.database	name of database connected to

%sql.error	last sql error received
%sql.insertid	id of last insert operation
%sql.connect	set when connected to a database

A special set of variables may be created by the application program under the name %global. %global.yyy variables are shared and may be accessed directly between all running script instances in Bayonne. The values stored in a global are also persistent for the duration of the server running.

In addition, DSO based functions will create variables for storing results under the name of the function call, and the DSO lookup module will create a %lookup object to contain lookup results. Also, server initiated scripts can pass and initialize variable arguments. For example, the fifo "start" command may be passed command line arguments, and these arguments then appear as initialized constants when the new script session is started.

6.1 Symbol Types and Properties

While most symbols are stored as fixed length strings, there are a number of special types that exist. "typed" symbols are not typed in the same way traditional typing occurs, however. "typed" symbols are typically symbols that perform automatic operations each time they are referenced as a command argument. Typed symbols may also format their content in specific ways. For example, the "date" symbol typpe always stores and presents dates in iso format ('yyyy-mm-dd'), even when set with date information in other formats.

Some symbol types are built into ccScript2 itself. Other symbol types are loaded through plugins with the "use" command. A few special symbol types are also defined by the Bayonne server itself and act like "built-in" types. The built-in types include char, string, number, integer, counter, sequence, fifo, array, lock, and stack.

By default new symbols are normally created as a fixed size "string". Many of the other types use this as well. A char symbol is a special case string symbol which holds only one character element. Numbers are also strings which can either contain integer values or fixed point decimal numbers.

The “counter” symbol is used as a typed symbol that automatically increments itself each time it is referenced. This can be useful for creating loop or error retry counters.

The “stack”, “sequence”, and “fifo” are symbols that can hold multiple values. A stack releases values each time it is referenced in lifo order until it is empty again. A fifo does this in fifo order. The sequence object repeats its contents when reaching the end of it’s list. Values are inserted into each of these types by using the special “post” keyword. “stack” and “fifo” may be used as global. objects to create and script ACD-like functionality in Bayonne.

A special type exists known as “array”. An array is treated internally much like a stack, a sequence, or a cache. The principle difference, however, is that the same value location is always referenced when the array is accessed, and this is known as the array “index”. The index is like a bookmark. It may be changed explicitly to point to a new entry with the “index” keyword, or may be changed automatically when examining an array through “foreach”.

When values are stored into an array, the index is automatically incremented to the next array element. Each array has a fixed maximum size when it is created. However, each array also keeps track of the last element actually stored, and this is known as the hi water mark. Hence, when performing a “foreach” operation, only those range of values from the start of the array to the highest element actually set will be examined, and not any unused elements to the very end.

The Bayonne server introduces two special symbol types; “timeslot” and “position”. A symbol that is defined as a timeslot can only be set or contain a numeric value which references a valid server timeslot. This can be used to verify if a session id is still valid, or to reference other sessions. The “position” type is used for using and manipulating audio timestamps. Audio timestamps are always returned in the format ’hh:mm:ss.nnn’. The “#” eval will return the timestamp value in seconds for convenience.

The time module introduces two special symbols, “date” and “time”. These provide symbols which always return an iso date and a standardized 24 hour iso time string. The “#” eval of a date symbol returns a julian number which can be used to calculate dates, and assumes a non-date formated number is a julian value. The “#” eval of a time symbol returns the number of seconds since midnight.

Other plugins may create additional symbol types in the future.

6.2 Session Ids and References

Session ids are symbol values that are used to refer to a Bayonne port that is running a call script. These references are used so that scripts in one port can, when needed, identify and reference scripts running in another port. The most common example of this is the start command, which can return a variable that will hold the session id of the script and port that a script was started on. This can then be used to rendezvous two script sessions for a "join". Similarly, the child script could also examine its %session.pid to find out who started it to join.

The most basic reference id is simply a port number. This reference is a numeric id, and is the same as the value that %driver.id returns. The disadvantage of using port numbers is that they are not aware of call sessions. Imagine I start a script on another port, say "3", and I decide to join to it later. In the interum, the call that was originally started has disconnected, and an entirely new call has appeared on port 3. I do not want to join to this new and unrelated call.

The second form of a session id is 14 bytes long. It is composed of a "-", a port number, another "-", and a call unique timestamp. This is known as a local session id. Many generated ids, such as those stored in %session.pickupid and %session.joinid, are passed in this form. This assures that the reference is not just to a specific port, but also to a specific call that is occuring on that port.

The third form of a session id starts with a node name and a "-", followed by a local session id. This form is considered unique for all call sessions on all server instances, assuming each server has a unique call node. These are best used when resolving activities that will be spread over multiple servers, such as call detail that may be collected into a single database.

All forms of session id references are supported by the built-in timeslot type. Usually one would declare a symbol as a "timeslot %myvar" and then use it to store the result of a child id, such as from a start or connect command.

7 Events

The event flag is used to notify where a branch point for a given event occurs while the current script is executing. Events can be receipt of DTMF digits in a menu, a call disconnecting, etc. The script will immediately branch to an event handler designated line when in the "top" part of the script, but will not repeatedly branch from one event handler to another; most event handlers will block while an event handler is active.

The exception to this rule is hangup and error events. These cannot be blocked, and will always execute except from within the event handlers for hangup and/or error themselves. Event handlers can be thought of as being like "soft" signals.

In addition to marking script locations, the script "event mask" for the current line can also be modified. When the event mask is modified, that script statement may be set to ignore or process an event that may occur.

The following event identifiers are considered "standard" for Bayonne:

identifier	default	description
`^hangup` or `^exit`	detach	the calling party has disconnected
`^error`	advance	a script error is being reported
`^dtmf`	–	any unclaimed dtmf events
`^timeout`	advance	timed operation timed out
`^0 to 9, a to d`	–	dtmf digits
`^pound or \verbŝtar=`	–	dtmf "#" or "*" key hit
`^ring`	–	ring event notification
`^tone`	–	tone event heard on line
`^wink`	–	wink line event
`^join`	–	session waiting for join
`^part` or `^cancel`	detach	conference/join disconnected.
`^fail` or `^invalid`	advance	failed process
`^event`	–	event message received
`^child`	–	notify child exiting
`^pickup`	–	we are picked up by another session

Some of these script events also have Bayonne variables which are set when they occur. When an event occurs and there is no handler present, very often execution

simply continues on the next statement, but the variable that is set may still be examined. The following event related symbols may be referenced:

%script.error	last script "error" message.
%pstn.tone	name of last telephone tone received.
%session.eventsenderid	trunk port that sent an `^event` to us.
%session.eventsendermsg	event message that is being sent.
%session.joinid	trunk port we last joined with.
%session.notifytext	text of unsolicited external notify
%session.notifytype	short description of external notify

8 Named Events

Bayonne also supports the use of arbitrary "named" event references as well as fixed x̂xx handlers in scripting. Named event handlers allows one to define an arbitrary named event in Bayonne and to offer an optional script handler to receive the event when Bayonne requests the event.

Named events are designated differently than traditonal handlers. They may use either the symbol, or may be enclosed in a pair of 's. The latter might be used if the named event contains spaces or other non-parsable characters.

Each named event is typically composed of a two part name. The first part represents the event classification, and the second part is the actual event. A ":" is used to seperate the two parts. Hence, a typical named event handler tag will look like xxx:yyy.

Named events may be used much the same way that traditional event handlers are used. They may be listed together and or'd, or even listed together with traditional event handlers. The latter allows named events to substitute for dtmf keyboard navigation, and allows, for example, the asr system to bind words to dtmf navigation. Consider, for example:

```
...
say "press 1 for yes or two for no"
listen count=1 timeout=30s

^1
@asr:yes
redirect ::doit

^2
@asr:no
```

```
redirect ::dont
```

In this case, the asr word recognizer is pushed from the listen command as descreat words which are directly associated with dtmf digit events. If either the key is pressed or the word is recognized, the event handler is requested. Furthermore, named events that appear after a x̂xx handler also inherit the same signal masks as the x̂xx handler.
The following named events are commonly used:

caller:xxx	caller number pattern to match
dialed:xxx	dialed number pattern to match
digits:xxx	digit pattern to match
digits:expired	collect command timed out
digits:partial	digits partially matched a pattern
digits:invalid	digits failed to match any pattern
dial:failed	sit tone or other failure to dial
dial:invalid	dialing reorder tone or invalid number
dial:noanswer	destination never ended ringback
dial:busy	destination busy
start:invalid	start destination number invalid
start:failed	start fails or dialing fails
start:expired	timeout without answer or join
start:busy	start for a busy number
start:running	child script running
seize:invalid	seized line invalid number
seize:busy	seized line busy number
seize:noanswer	seized line number not answering
seize:failed	seizure failed
input:timeout	timed out waiting for input
input:invalid	invalid input
menukey:flash	line flash as menu event
menukey:none	no menu key entered
menukey:invalid	invalid menu key used
menukey:x	entry for specific menu key digit
menukey:timeout	timed wait for menu key timed out
playkey:x	replay command menu key event
recordkey:x	record command menu key event
exitkey:x	input command exit key event
line:wink	line wink event
line:disconnect	line loop drop

incoming:ringing	pickup for ringing line
incoming:forward	pickup for forwarded call
incoming:recall	pickup for recalled call
incoming:pickup	direct station pickup
incoming:immediate	immediate pickup
tone:xxx	some tone event detected
error:xxx	a specific error event occurred
event:text	an inter-port send text message event
parted:xxx	join parted reason event (to be added)
answer:failed	answer wait failed

Of all the named events, "digits:" events are treated very special. These are used to impliment dialing plans. The actual digits are matched from The match can be done not just by absolute values but also by patterns. Several special characters are reserved for this:
The "N" character is a place holder for any value of 2-9, and the "X" character is a placeholder for any decimal value. In addition, the letter "O" can be used for a lead one which is ignored if not present, and "Z" may be used for a lead zero which is also ignored if missing. Finally, the digit pattern can be made to only be in effect for a specific country location, using a countrycode/. The effective country code is found in %session.dialing. For example, a PBX application which needs to dial numbers for north american dialing plan when used in the US, and for different numbers in "country 389" (Macedonia) could be represented, with lead '9' for outside line select, with 911 service if only in US, and with Macedonian Emergency numbers, as follows:

```
@digits:389/92 # police
@digits:389/93 # fire
@digits:389/94 # ambulence
@digits:1/911 # quick shortcut
@digits:1/9911 # pbx dial "9" first mode...
redirect ::emergency

@digits:389/99
@digits:1/011
@digits:1/9011
redirect ::international

# outside number in north american dialing plan
@digits:1/90NXXNXXXXXX
# Macedonian cities are 9x followed by 6 digits?
```

```
@digits:389/9XXXXXXX
redirect ::outsidedial
```

This allows one to create scripts that can be localized easily, even with very complex dialing plans. The digits: events can be used during any dtmf input, including collect. The route command can be used to perform arbitrary digit routing and pattern matching based on the value of a symbol.

9 Loops and conditionals

Scripts can be broken down into blocks of conditional code. To support this, we have both if-then-else-endif constructs, and case blocks. In addition, blocks of code can be enclosed in loops, and the loops themselves can be controlled by conditionals.

All conditional statements use one of two forms; either two arguments seperated by a conditional test operator, or a test condition and a single argument. Multiple conditions can be chained together with the "and" and "or" keyword.

Conditional operators include = (or -eq) and <> (or -ne), which provide integer comparison of two arguments, along with >, <, <=, and >=, which also perform comparison of integer values. A simple conditional expression of this form might be something like `if %val < 3 ::exit`, which tests to see if %val is less than 3, and if so, branches to ::exit.

Conditional operators also include string comparisons. These differ in that they do not operate on the interger value of a string, but on it's effective sort order. The most basic string operators include == (or .eq.) and != (or .ne.) which test if two arguments are equal or not. These comparisons are done case insensitive, hence "th" will be the same as "Th".

A special operator, "$", can be used to determine if one substring is contained within another string. This can be used to see if the first argument is contained in the second. For example, a test like "th $ this" would be true, since "th" is in "this". Similar to perl, the " " operator may also be used. This will test if a regular expression can be matched with the contents of an argument. To quickly test the prefix or suffix of a string, there is a special $< and $> operator. These check if the argument is contained either at the start or the end of the second argument.

In addition to the conditional operators, variables may be used in special conditional tests. These tests are named -xxx, where "-xxx argument" will check if the argument meets the specified condition, and "!xxx argument", where the

argument will be tested to not meed the condition. The following conditional tests are supported:

conditional	**description**
-defined	tests if a given argument is a defined variable
-empty	tests if the argument or variable is empty or not
-script	tests if a given script label is defined
-module	tests if a specific .use module is loaded
-voice	tests if a given voice exists in prompt directory
-lang	tests if a given language module is installed
-timeslot	tests if a given argument is a valid timeslot
-driver	tests if a specified driver is installed
-span	tests if a specified span exists
-file	tests if a specified prompt file exists
-dir	tests if a specified directory exists
-key	tests if a specific persistent key value exists
-localtime	used to create scheduler tests with time module

The “if” expression can take three forms. It can be used as a “if ...expr... label”, where a branch occurs when an if expression is true. it can be in the form “if ...expr...” followed by a “then” command on the following line. The then block continues until an “endif” command, and may support an “else” option as well. This form is similar to the bash shell if-then-fi conditional. Finally, if the conditional is needed for only one statement, there is a special case form that can be entered on a single line, in the form “if ...expr.. then command [args]”, which allows a single statement to be conditional on the expression.
The “case” statement is followed immediately by a conditional expression, and can be used multiple times to break a group of lines up until the “endcase” is used or a loop exits. The “otherwise” keyword is the same as the default case in C. A set of “case” expressions and “otherwise” may be enclosed in a “do-loop” to get behavior similar to C switch blocks.
The “do” and “loop” statements each support a conditional expression. A conditional can hence be tested for at both the top and bottom of a loop. The “break” and “continue” statements can also include a conditional expression.
In addition to “do-loop” there is “for-loop”. The “for-loop” can take several forms, as noted by “for” and “forech”. “foreach” to process a packed list, to iterate the contents of an array, or extract and process the contents of a stack or queue. The default “for” statement assigns a variable from a list of arguments, much like how for works in bash. In all cases, break and continue can still be used within the loop.

10 Subroutines and symbol scope

Bayonne recognizes the idea of symbol scope. Symbol scope occurs when refering to variables that are either "global" in scope, and hence universally accessable, or that are "local". Local symbols exist on a special heap, and a new local heap is created when a subroutine level is called.
Global scope symbols are those that have componentized names. Hence "%xxx-.yyy" is treated as a global symbol. Local symbols do not have componentized names. Hence "%yyy" is a local scope symbol. This allows one to determine scope purely from symbol name, rather than requiring implicit commands to create a symbol that is local or global.
Bayonne scripting recognizes subroutines as a label that is invoked through "go-sub", or a "function" that is called. When a "gosub" or "call" statement is used, execution is transfered to the given script, as a subroutine, and that script can then return to continue script flow with a "return" statement.
When invoking a subroutine or a function, a new local variable heap will be created. Local variables are then created in the context of the subroutine only, and any changes are lost when "return" is used to return back to the calling script. The "return" statement can be used to transfer values, typically from a subroutine or function's local heap, back to a variable in the calling script's context. This is done with "var=value" lists that may follow the return statement, as in "return status=ok", for example.
Subroutines and functions may also be invoked with paramatric paramaters. These paramaters are then inserted into the local heap of the newly called subroutine and become accessible as local variables. This also is done with keyword value pairs, as in "call ::myfunc myvar=3", for example. When this is done, a local constant is created, known as %myvar, that is then usable from ::myfunc, and exists until ::myfunc returns. Since this is a constant, its value may not be altered within ::myfunc.
Sometimes a subroutine can contain initialization values to use if no parametric value have been passed. Since parametric values are constants, they cannot be altered, and hence, one can do something like:

```
protected mysub
        const %myvar 4
        ...
```

And thereby define %myvar locally as 4, unless there was a "call ::mysub" with an alternate value being passed as a myvar=xxx.
Subroutines and functions also support call by reference. This can be used to permit a subroutine to directly modify a local variable in the scope of the calling

script automatically. Call by reference is done by using a keyword=&var form of keyword. Consider the example:

```
...
        set %mysym "test"
        call ::myfunct myref=&mysym
        slog %mysym
...

function myfunc
        set %myref "tested"
        return
```

In this case, the slog will show "tested" since %myref in ::myfunc actually points back to %mysym in the calling script.
Subroutines differ from functions in several ways. First, when a subroutine is called through "gosub", the event handlers of the calling scripting script are ignored, where a function enables inhereting of event handlers. It is possible to call a subroutine that uses the same local variables directly, rather than having it create a new local context. This can be done using the "source" statement, or when using "import" to import a script module. The "import" statement not only imports the named script module at compile time, but also, at runtime, invokes a "source" statement of the code at the front of the script module.

11 Keywords and Reference Scope

Keywords are used in several forms in Bayonne scripting. Those most common form is as an associative or keyword option to a given command. For example, the "set" command may have a "size=" keyword, which can be used to specify the "size" of new variables being created. User defined functions also operate this way by creating a local instance of the variable being used as a keyword. Hence, a "call" statement to a function with a "size=" option will create a local scope constant "argument.
Basic keywords can either have a single string constant or a variable. Hence, the "size=" option may have a single string constant like size="10", or a reference to another variable, as in "size=%mysize" or "size=.mysize" to reference a current local or global scope variable by value, or "size=&mysize" or "size=&.mysize" to reference the same by name. Complex or compound string expressions, such as

those that may contain variables, may NOT be used in string constants passed as command arguments, as they will not be parsed and substituted for their variable values.
Some commands use keywords for other purposes. The “return” command uses a keyword to refer to variables in the context of the calling script invoking a function. When keywords are used to refer to variables, the various forms of variable reference by scope may also be used. Hence, to set a local variable in the called context on return, one might use “return status=1” or “return %status=1”. One may also refer to variables in global scope, such as “return %session.digits=“1”” or “return .myvar=7”.
Some commands use keywords to refer to content that is being saved into another variable. This can be expressed either as a “=¿” or a “=&” keyword. For example, to save the file count of the “list” command into a script variable, one might use “list count=¿.mycount” or “list count=&.mycount”. The “=¿” form is prefered as it makes it clear in reading what is actually happening.
Finally, the SQL module may use keywords to refer to or bind column names to script variables. When this is done, the keyword is preceeded with a & and refers to a database column that will be stored into or set from scripting, as in “sql.fetch &name=¿.myvar” for example.

12 Transaction Blocks

In addition to functions, subroutines, loops, and conditional statements, scripts may be gathered together under transaction blocks. Normally each script statement is step executed over a timed interval. This is done to reduce loading when deriving several hundred instances of Bayonne for a high density voice solution. However, some scripts either involve statements that are trivial or that need to be executed together. These can be done using a “begin” and “end” block.
When a transaction block is used, all the statements within it are executed as quickly as possible as if they were a single script step. This allows one to go through a series of set or const definitions quickly, for example.
In addition, “begin” may be used in front of a cascading case block, or before an “if” statement. This allows all the conditional tests within the case or if to be executed together until the “endif” or “endcase”, rather than depending on stepping.
Transaction blocks only work for statements that do not involve scheduled operations. Things that schedule include sleep, playing and recording of audio, and libexec statements. When these appear within a transaction block, the transaction block is suspended for those specific statements, and then resumes on the next unscheduled statement.

Transaction blocks will automatically exit when a branch statement is enountered, or when a “end”, “endif”, “loop”, or “endcase” is encountered. Transaction blocks cannot encapsulate a loop, and they will not operate with a subroutine call since calling a subroutine is a branching operation.
When a “function” is called, it is automatically invoked as a transaction block. Hence, you never need to use a “begin” statement at the start of a function to improve function call performance. An event handler that includes just a “goto” or “return” statement (as well as when such a handler starts with a “begin” statement) is also executed immediately when the triggering event is received. Finally, there is tunable support for executing X number of non-branching consequetive statements together automatically without requring “begin” as part of the normal runtime environment. These things often make the use of explicit “begin” statements unessisary.

13 Files, paths, prompts, extensions, and directories

Prior to Bayonne scripting rev II and III, there used to be all kinds of complicated rules and special prefix values that effect how and where audio prompts are stored and played, and some would depend on the name of script file itself. As of rev III, there is only a voice library, and a default prompt directory for non-voice / non-language localized pre-generated prompts. Dynamic prompts are still stored and organized in subdirectories on the /var/lib/bayonne path.
To refer to an audio sample stored in a var path subdirectory, all one needs to do is refer to the partial path as a filename, as in xxx/yyy.au. The play and record command, and others, also support a “prefix=” option, which can be used to specify the xxx subdirectory name seperately. This may be convenient when audio prompt filenames would need to be otherwise constructed from concatenated symbols or strings. The special compile time directive, **dir** may be used to assure that the xxx subdirectory exists in /var/lib/bayonne. Hence, if we wish to record messages into /var/lib/bayonne/msgs, we can use either something like “record prefix=msgs myfile” or “record msgs/myfile”.
For audio that is stored in /var/lib/bayonne, a file extension is automatically added if none is specified. This default file extension is set in the [script] section of bayonne.conf, and is stored in %audio.extension. The default extension is typically set to .au, for sun audio files. For example, to record msgs/myfile as a .wav file, this could be done with “record prefix=msgs extension={.wav} myfile”. The extension can also be specified directly as part of the filename, as in “record msgs/myfile.wav”.

When using the "prefix=" modifier, there is a special prefix with a pre-reserved meaning. When using "prefix=memory", rather then using audio for /var/lib/bayonne/memory, the audio files are actually stored on the tmpfs, in ram (or swap), which usually is /dev/shm. This can be used to record special prompts that maybe need to be processed further. It is particularly useful when constructing a rotating "feed" which may be recorded from one channel and listened to in realtime by multiple callers on other channels.
When using simple filenames, as as "test" or "1" in "play" or "speak" commands, a very simple set of rules is followed. If the file can be found in the current voice library, it is played from there. For a typical install, a voice library might be /usr/share/bayonne/audio/en/male, or the default prompt library, /usr/share/bayonne/audio/none/prompts. The prefix directory for all voice libraries, and the path for the default prompt library are specified in the [paths] sections of bayonne server.conf, and may be overriden in virtual hosts through virtual.conf. The voice library will include files converted to several audio formats, and these will be chosen by the driver on a per port basis as needed. The non-voice prompt files are stored under the default file extensions and formats specified for prompts stored in the var path.
In addition to standard URL's and voices, there are a number of special pseudo-urls's that have special meanings. These are defined below:
mem:
This has the same effect as using the "prefix=memory" option, and refers to audio stored or retrieved from system memory as organized by the tmpfs. The tmpfs usually is mounted as /dev/shm.

tmp:
This refers to audio that is stored (or recorded to) through the /tmp filesystem. This allows use of /tmp for audio storage.

xxxx:
Any reference to a url that does not use any of the special keywords is used to refer directly to a subdirectory of the voice library directory. In addition to referencing alternate languages in the middle of a phrase, this can be used to organize system audio prompts used in scripting. The actual location is found from /usr[/local]/bayonne/audio/none/xxxx/.

14 Reserved names and namespace collisions

Bayonne uses a simple namespace architecture. A given script file (.scr) or macro (.mac) is compiled into a named script. Each section or function is given a xxx::

scrope, based on the xxx.scr or xxx.mac filename, followed by the name of the section or function. Obviously macros and script files of the same name cannot be loaded together.
In addition to macros, some script file names have special meanings in Bayonne namespace, and also cannot be used. The following table lists and describes each of these for both Bayonne and Troll type servers:

exec::xxx	formed from bayonne wrappers
libexec::xxx	formed from .apps wrappers
caller::xxx	troll assignment caller.conf
dialed::xxx	troll assignment dialed.conf
slot::xxx	troll assignment slots.conf
span::xxx	troll assignment spans.conf
route::xxx	troll routing gateway.conf
ppp::xxx	troll protocols (sip.conf, etc)

15 Compile Time Effects

In Bayonne Rev III scripting, commands can have compile time effects only, and examples of this include such things as "use" and "assign". Compile time commands look like ordinary script commands, but they only perform an operation as a script is being compiled, and are not represented in the runtime script image produced by the compiler.
Commands can also have runtime effects only, and most commands fall into this category. That is, they perform operations when the script command is executed during the course of Bayonne call processing. Some commands can have both compile time and runtime effects. In addition to compile time commands, we have compile time symbol substituon. Compile time symbols are given a value at the time the script is compiled, and may be considered like a constant.
We also have three special compile time symbols to help with debugging. These are "$script.name", "$script.file", and "$script.line". The first refers to the name code is currently being compiled under. When exporting, this may in fact be different than the filename, which may be found with "$script.file". The final symbol expands to the current statement line number being compiled.
The following compile time directives exist:
import — requires *filename ...*
Requests compilation of a Bayonne script module in your default or current virtual session. Once compiled, the script module's functions may be referenced. "import" also functions as a "source" statement at runtime to run the start of the imported module. This is often used to initialize symbols that the modules functions will then depend on.

dir *subdir ...*
Create the specified subdirectory in /var/lib/bayonne if it does not exist because the current script requires it to store audio data. **load** *module ...*
Load a Bayonne 2 management plugin.

lang *module ...*
Load a Bayonne 2 phrasebook module plugin. Needed for voicelib command and voices= options to work.

use *packagename ...*
Install a bayonne script plugin module that the current script requires, if it is not already installed.

assign[**.mode**] *values ...*
Assign the currently compiling script, whether to answer on a time slot or for a span, or an incoming did or calling number. The mode may also be a driver, which invokes a driver specific assignment routine.

assign.did—**assign.cid** *values ...*
This is used to assign incoming pstn numbers dialed to us, or the telephone number of a specific caller to a script. These used fixed numbers, without patterns.

assign.span—**assign.timeslot** *values ...*
You can assign physical timeslots or spans to be answered by a specific script when a call arrives. This is only meaningful for pstn cards. You can use a value that includes a relative offset for a driver, or for a timeslot, relative offset of a span, since you may not know what timeslot your driver was initially starting at.

assign.sip *ipaddr—myuri@myhost ...*
This is used to assign scripts for anonymous sip connections. You can either publish a local uri that will run the assigned script when invited, or allow all incoming connections from a specific proxy server to anonymously peer with us.

caller *number ...*
This is an alias for assign.cid.

incoming *number ...*
This is an alias for assign.did.

register[**.driver**] *value ... [options=values]*
Register the currently compiled script with a proxy server or gateway through the specified driver. This is typically used to register scripts with a SIP proxy server.

register.sip *[type=mode] [realm=realm] [server=proxy] [user=userid] [secret=password]*
This is used to register a script with a sip proxy. The type controls how incoming connections are handled. For type "peer" or "friend", incoming connections can invoke arbitrary uri's which become the %session.dialed for the script. For type "user", the script is ran directly without dialing plan selection in all cases. With exosip2 we will add type "client" and use "peer" to allow remote endpoints to authenticate with us.

16 Database Connectivity

In Bayonne, database support is supplied through a special driver plugin. The database driver plugin provides an interface to the Bayonne "sql" command. The Bayonne "sql" command can then be used to build and issue sql queries, as well as to fetch the results of queries. For Rev II of Bayonne, we looked at how to simplify the construction and use of sql databases queries as much as possible, as well as how to integrate database support more deeply into Bayonne scripting.
You can still only have one database plugin active for your server and can only connect to the tables of a single database. However, while each virtual host also uses the same plugin, each virtual host can be connected to a seperate database (dsn). Hence, if you use the unixODBC plugin, virtual sessions can connect to different underlying databases through the dsn, and this provides one means to supporting multiple database "drivers" without needing to do so in Bayonne directly.
When Bayonne starts as root, it changes it's effective user id to that of the "bayonne" user, as specified in /etc/bayonne/server.conf. This also means that Bayonne will connect to databases under the "bayonne" user when started under the root account. For things like the unixODBC driver, the data source connections should be defined under the "bayonne" user account, and access should be assumed to occur under "bayonne".
When database connections need to be authenticated, the authentication information is stored as options to the "use odbc" command used to install the database plugin. Thee authentication information stored there should also be that relevent to connecting as a "bayonne" user, and a "bayonne" user should be added to the list of database users.
Each script that wishes to make a database connection has a private and seperarte session with the database, as defined by the use of a "use odbc" command at compile time. Hence multiple script files can be written and each can have a session and connection to a seperate or different database.
When a sql command is issued in each script that has a database, Bayonne establishes a connection to the database. This connection is kept persistent until

the time the call completes. If additional calls also happen to make sql queries, they share the same connection rather than having to establish a new one, and the connection is not torn down until the very last call using the database terminates. This allows for stable connection pooling and persistant high for performance.
To issue a query, one just uses the "sql" command followed by the arguments used to build up the query string. For example, to select a row of pin numbers under a common account, one might do something like "sql "SELECT pin, name FROM mypins WHERE account="" %myaccount """". Doing this one can build sql statements that include values from Bayonne scripting variables, which often would be used for where classes.
For sql commands, such as "select", which may return multi-row results, the results can be stored directly into a ccScript array using the "results=¿varname" option. The array can then be iterated with the "foreach" operator as needed. If a single row is returned, an ordinary script symbol can be used to save the results.
Another special sql command is "sql.header". This command can be used to retreive the column names of the last query into ccscript variables. This can be used similar to "sql.fetch". For example, one can use "sql.header %col1 %col2".

17 Command Reference

17.1 Variable declarations

These commands describe the various means to create or initialize a symbols in the scripting language. Symbols may be of specialized types that automatically perform operations when referenced, or generic symbols of a specific type or size.

array[**.count**] *%var(s)...*
array *size=size count=count %var(s) ...]*
Create a Bayonne array composite object. The object may have values stored in it, and may be selectivily accessed through the use of the "index" keyword or iterated through a "foreach" loop.

char *%var ...*
Create single byte string variables.

clear *%var ...*
Clear (empty) one or more variables. It does not de-allocate. This means that if you need to determine whether a variable is "there" or not in a TGI script which

is passed the variable, the empty string is equivalent to a nonexistent variable. For arrays, this command clears the array content and resets the hiwater mark to the start of the array. For special variables, rather than being cleared, a default value may be set. For example, clearing a date variable will set it with the current date.

const *%var values...*
Set a constant which may not be altered later. Alternately multiple constants may be initialized.

counter *%var*
Create a variable that automatically increments as it is referenced.

cat *%var values...*
Append additional data to an existing variable.

date *%var*
Create a variable to hold ISO date values. All values set into the variable are converted to iso date format automatically. The variable is initialized to the current date. Requires "use time".

fifo[**.count**] *count %var(s)...*
fifo *size=size count=count %var(s)...*
Create a ccScript fifo "stack" variable. This creates a variable that automatically unwinds from first in to last in when referenced.

init *%var values...*
Initialize a new system variable with default values. If the variable already exists, it is skipped. Optionally multiple variables may be initialized at one.

integer *%var ...*
Create numeric variables which hold integer numbers.

number[**.decimal**] *%var ...*
Create variables which hold fixed point decimal numbers. Different decimal sizes are rounded to match the variable being stored into.

position *%var ...*
Create variables to hold millisecond audio timestamp positions.

stack[**.count**] *count %var(s)...*
stack *size=size count=count %var(s)...*
Create a ccScript fifo "stack" variable. This creates objects that unwind auto-

matically on reference.

ref *%ref components ...*
This can be used by a subroutine to form a local instance of a reference object that points to a real object in the public name space.

set *%var values...*
Set a variable to a known value. If the variable already exists, it is changed to the new value.

sequence *%var value(s) ...*
Create a symbol which, when referenced, emits the next value of a sequence of values.

string[.size] *[size=bytes]* ***var[.size]*** *%var(s)...*
Pre-allocate "space" bytes for the following variables. If no size is given, the default size (64 bytes) is used.

time *%var ...*
Create variables to hold time of day. The variable is initialized to the current time of day. Requires "use time".

timeslot *%var ...*
Create variables which holds valid Bayonne timeslot references.

17.2 Symbol manipulation

While ***set*** *is perhaps the most common way to both define and manipulate a symbol, there are a number of additional script commands that can be used for this purpose.*

expr[.decimal] *%numvar expression...*
expr[.type] *%typevar expression...*
*Expression is used to set a symbol to the result of a numeric expression. Expressions may be paranthetic, and use the the +, -, *, /, and mod (set a type based on a numeric expression. For example, an expression could be used to set a date variable based on an expression manipulating julian values.*

first *%var value ... [field=pos] [token=char]*
Remove a specific value entry from a stack or fifo, and then array variable and then add the new value or packed list as the first entry in the stack or fifo. The

specific entry can be found as a member of a packed list as defined by a token and field offset.

index *value %arrays...*
This is used to set the current index value of a given array so that a specific element in the array may be accessed, and all values set into the array start from the given index.

last *%var value ... [field=pos] [token=char]*
Remove a specific value entry from a stack or fifo, and then array variable and then add the new value or packed list as the last entry in the stack or fifo. The specific entry can be found as a member of a packed list as defined by a token and field offset.

pack *%var values ... [prefix=s̈tring]̈ [quote=char] [suffix=s̈tring]̈ [size=bytes] [field=offset]*
Pack a variable from a list of values. The contents are seperated by the pack token, and may be quoted. Quoted contents are packed as a comma (or token) delimited string.

remove *%var value ... [field=pos] [token=char]*
Remove a specific value entry from a stack, fifo, sequence, or array variable. The specific entry can be found as a member of a packed list as defined by a token and field indicator. Hence, a stack of packed lists which include some id code field could be searched for and stripped.

unpack *%var %vars... [size=bytes] [field=pos] [token=char] [quote=char]*
Unpack a packed variable into its constituent elements either into existing variables, or new ones of specifized size, based on the token and/or qouting. For example, this can be used to decompose an iso date into a year, month, and day variable by using as the token.

17.3 Script and Execution Manipulation

There are a special category of script commands that directly deal with and manipulate the script images in active memory. These include commands that operate on scripts by creating local template headers, or user session specific copies, of loaded scripts, which can then be selectivily modified in some manner. The following script manipulation commands exist:

form *[initializer list]*

endform *[loop condition]*
Form blocks are used to collect together a sequence of prompts and input or keyinput statements which can be used as an interactive form. Prompt commands stop playing once input has been received, and the input or keyinput statements can be used at the end to find what the user has input as well as wait for completion of input. Forms can have initializers and optional looping conditions.

begin
end
Mark the start or end of a script transaction block.

endinput
Used to disable further input in the current form or subroutine.

gather *%var suffix*
Gather the number of instances of a given xxx::suffix scripts that are found in the current compiled image, and store the list of named scripts in the specified %var.

session *[id]*
Select the session of another active instance of Bayonne and access non-local variable content from that session. Use session by itself to restore the original running session.

lock *%var*
Create a symbol or use an already created symbol that is a lock type variable, and lock it for the current running session. The value contained can be tested afterward to see if you have the lock, or if another session already locked it. The lock can be removed with the "clear" command.

17.4 Looping, Branching, and Conditionals

break *[value op value][and — or ...]*
Break out of a loop. Can optionally have a conditional test (see if).

call *function [var=value ...]*
gosub *label [var=value ..]*
Call a named function or subroutine with optional arguments. When the function or subroutine completes, control is returned to the calling script on the next statement.

continue *[value op value][and — or ...]*

Continue a loop immediately. Can optionally have a conditional test (see if).

case *[value op value][and — or ...]*
otherwise
endcase
The case statement is a multiline conditional branch. A single case (or ***otherwise****) line can be entered based on a list of separate case identifiers within a given do or for loop. The* ***otherwise*** *statement is used to mark the default entry of a case block.*

do *[value op value]*
Start of a loop. Can optionally have a conditional test (see if). A do loop may include ***case*** *statements.*

exit *[exit-code]*
Terminate script interpreter or invoke the special "::exit" label in the current script. If the script is called from an external script or service service, then a exit result may also be posted back to the external script. If the script was started by another port, then that port will receive a ĉhild event and also receive the exit-code specified.

for *%var values...*
Assign %var to a list of values within a loop. A for loop may include ***case*** *statements. For is similar to it's behavior in "bash".*

foreach *%array [index=[&]startindex]*
foreach *%var %packedvar [token=char] [index=[&]startindex*
Automatically set the "index" of the named array to the next element starting from 1, and process a loop. The loop can then refer the the array and examine each consequtive element in it. An optional starting index can be passed. If the starting index is a &var reference, then the current index value will also be saved into the variable. Foreach can also be used to process and extract the contents of a packed variable.

goto *label*
Goto a named script in a script.

if *value op value [and — or ...] label*
Used to test two values or variables against each other and branch when the expression is found true. There are both "string" equity and "value" equity operations, as well as substring tests, etc. Multiple conditions may be chained together using either ***and*** *or* ***or****. In addition to simple values, ()'s may be used to enclose simple integer expressions, the results of which may be compared with operators*

as well.

if *value op value [and — or ...]* ***then*** *[script command]*
Used to test two values or variables against each other and execute a single statement if the expression is true. Multiple conditions may be chained together using either ***and*** *or* ***or****. In addition to simple values, ()'s may be used to enclose simple integer expressions, the results of which may be compared with operators as well.*

if *value op value [and — or ...]*
then
else
endif
Used to test two values or variables against each other and start a multi-line conditional block. This block is enclosed in a "then" line and completed with a "endif" line. A "else" statement line may exist in between.

loop *[value op value][and — or ...]*
Continuation of a for or do loop. Can optionally have a conditional test (see if).

repeat *count*
Repeat a loop for the specified count number of times.

restart
Restart the current script from the top.

return *[label] variable=value ...*
Return from a gosub. You can also return to a specific label or `^handler` *within the parent script you are returning to. In addition, you can use* ***return*** *to set specific variables to known values in the context of the returned script, as a means to pass values back when returning. Finally, the .exit option may be used to exit the script if there is no script to return to and .clear may be used to clear the dtmf input buffer.*

route[.name] *value*
Branch to a @digits:xxx pattern entry based on a value. If the .name option is used, then that name will be used in place of the default digits:.

signal *ĥandler*
Branch to a signal handler.

source *label*
Source is used to invoke a subroutine which uses the current stack context, and is somewhat similar in purpose and effect to "source" in the bash shell. "source"

is executed at runtime for a compile time "import" statement.

throw *event*
Select and seek a named event handler attached to the script to catch the throw.

17.5 Basic data sets and logging

Bayonne has some simple commands for logging.

echo *text...*
Echo output to stdout when in the foreground, or the .out logfile when in daemon mode.

slog *[.debug |.info |.notice |.warning |.err |.crit |.emergency] message...*
Post a message to the system log as a `syslog(3)` *message. The logging level can be specified as part of the command. If no logging level is specified the message will be logged as* `slog.notice`*.*

If Bayonne or the stand-alone ccScript interpreter are running on a console or under `initlog(8)`*, the messages will be output on the standard error descriptor, not standard output. Note that you cannot use % characters in the strings to be outputted.*

write *file text... [prefix=dir]*
Append text to an existing file by echoing it to the file. If you need to write to an empty file, then use erase first.

17.6 Bayonne input processing commands

These commands deal with proicessing dtmf digits which may be input by the user.

cleardigits *timeout=debounce*
Clear any pending input. The menudef command also clears any pending input. However, cleardigits can do so anywhere, and can include a timeout to wait for no input.

collect *digits [timeout [term [ignore]]]*
collect *[var=&sym] [count=digits] [exit=term] [ignore=ignore]*

Collect up to a specified number of DTMF digits from the user. A interdigit timeout is normally specified. In addition, certain digits can be listed as "terminating" digits (terminate input), and others can be "ignored". The .clear option can be used to clear the input buffer before collecting, otherwise any pending digits in the dtmf session buffer may be processed as input prior to waiting for additional digits. The .trim option can be used to strip out any additional digits that may still be in the buffer after collection count.

keyinput *%var [timeout=wait] [menu=keys]*
Read a single DTMF input key that may be waiting in the input buffer. If no input is ready, wait for the specified timeout. If keys are referenced in the menu keyword, then @menukey:x events will also be triggered.

input *%var [format=mask] [exit=keys] [timeout=timer] [interdigit=waitafterfirst] [count=size]*
Input can be used to collect and wait for DTMF input that is then saved into a variable. Input is usually assumed to stop either when a specified count of digits has occured, or an input end key has been pressed. The '#' key is the default end of input used if none is specified. Format masks can be used to format input for specific variables. For example, to input a iso date into an iso date variable one could use a format of ????????. The input is then split to fill ? fields.

read *%var [format=mask] [exit=keys] [timeout=timer] [interdigit=digits]*
Read is similar to input. However, it is normally used to input a fixed number of characters rather than a field with a terminated input. Finally, where input automatically clears the entire DTMF buffer after completion, read only removes those digits it processes from the input buffer.

17.7 Bayonne tone processing commands

tone *frequency=tonefreq [duration=timeout] [level=volume]*
tone *freq1=f1 freq2=f2 [duration=timeout] [level=volume]*
Generate a generic single or dual-tone sound for a specified duration.

reorder *[timeout=duration] [location=cc]*
Genorate a reorder tone.

dialtone *[timeout=duration] [location=cc]*
Generate a dialtone.

busytone *[timeout=duration] [location=cc]*

Genorate a busy tone.

ringback *[timeout=duration] [location=cc]*
Generate a line ringing tone.

beep
Generate a short beep, such as a lead in for record.

sit
Generate call intercept sequence.

callwait
Generate call waiting tone sequence.

callback
Generate callback tone sequence.

dtmf *digits interdigit=timeout*
Generate a DTMF dialing/tone sequence.

mf *digits*
Generate a MF dialing/tone sequence.

17.8 Bayonne call processing commands

This covers the basic set of call processing script command extensions that are common and are generally usable with all Bayonne drivers. Some of these commands may depend on specific bayonne plugins or extensions to be installed.

dial *[timeout=cptimeout] number...*
This performs dialing with something that is a standard international number, as in "+1 800 555 1212", for example. These can be in symbols, or other places, and are dialed on the public network as a network number through the driver. Normal numbers may also be passed, and either may appear in a symbol, as a literal, or composed from multiple values.

transfer *number*
Transfer the call to another location, and then hangup.

hangup
This is essentially the same as the ccScript "exit" command.

libexec[.timeout] *program [query-parms=value ...]*
Execute an external application or system script file thru the Bayonne TGI service. This can be used to run Perl scripts, shell scripts, etc. A timeout specifies how long to wait for the program to complete. A timeout of 0 can be used for starting "detached" commands. Optionally one can set libexec to execute only one instance within a given script, or use .play to run an external tgi that will generate an audio file which will then be played and removed automatically when the tgi exits.

play *[prefix=path] [voice=voicelib] [extension=fileextension] audiofile(s)*
Play one or more audio files in sequence to the user. Bayonne supports raw samples, ".au" samples, and ".wav" files. Different telephony cards support different codecs, so it's best to use ulaw/alaw files if you expect to use them on any Bayonne server. Optionally one can play any of the messages found, or only the first message found, or a temp file which is then erased after play. The play command recognizes and parses phrasebook statements as well as audio files.

speak *[prefix=path] [voice=voicelib] [extension=.ext promptfiles...*
This command is identical to play with one important exception. If it is used in a menudef/endmenu block, then if dtmf input is pending, the command is skipped. Similarly, if a key is pressed during play, the prompt command ends.

replay *[prefix=path] [extension=.ext] [offset=timestamp] [menu=keys] filename*
Replay is used to play a single file, and likely one that has been created through record, rather than from the prompt library. The menu option allows @playkey:x events to be attached which can be used to create menus for things like skip ahead, etc, to support navigation through the file. When ending, the %session.position holds the last timestamp, and this can be used to compute a new timestamp for skips if caused by menu keys.

record[.vox] *[prefix=[path] [offset=timestamp] [encoding=format] [duration=maxtime] [extension=.ext] [note=annotation] [silence=detecttime] filename [rename]*
append[.vox] *[prefix=[path] [extension=.ext] [silence=detect] [duration=maxtime] filename [rename]*
Record user audio to a file, up to a specified time limit, and support optional abort digits (DTMF). Optionally one can append to an existing file, or record into part of an existing file by offset. Record with save= option means the file is saved or moved to the specified name if recording is successful, replacing what was previously there. The rename option can be used to set automatic file rename at normal end of record or if caller hands up.

sleep *timeout*
Sleep a specified number of seconds or milliseconds.

sync *time*
Sleep until the specified number of seconds are reached in the call duration. This can be used to sync prompts or operations that might need to occur 30 seconds into a call, for example.

17.9 Bayonne session management commands

This covers Bayonne commands related to session management where multiple sessions are initiated and channels are interconnected. When connecting by "number", a number in uri form is recommended. Hence, for sip, you should use sip:xxx or sip:xxx@yyy, and for pstn, you should use pstn:xxx.

connect *number script [caller=num] [display="text"] [timeout=timer] [duration=jointime] [tone=ringback4]*
Initiate a connection to the specified telephone number through a named "selectable" script on a new session. Connect will then wait for the new session to join, at which point a full duplex connection is formed, with an optional duration. A tone can be specified to play while waiting for the join. ringback4 or ringback6 are suggested, as they have a lead silence which should be long enough to cover busy detection.

join
A selectable child script that has been started by a connect can use join to accept and complete the connection.

start *number script [caller=num] [display="text"*
Start a selectable script. Execution resumes immediately. The child script runs independently.

17.10 File command extensions

A number of commands are actually present in a special plugin known as "files". You can install these commands with the "use files" statement. Generally, these commands deal with files that exist in the var path.

build *[prefix=path] [file=target] [voice=voicelib] [language=translator] prompts...*
This command constructs a new audio output file using the phrasebook rules. This can be thought of as functionally similar to the "speak" command, but where output is used to create a new audio prompt rather than to be spoken immediately.

copy *[prefix=path] [file=target] [voice=voicelib] [extension=ext] [format=format] prompts...*
Copy a series of audio files or prompts into a new destination audio file on subdirectory of /var/lib/bayonne. The copy command processes filenames in a manner similar to play to reference audio from the voice library, and will do basic audio format conversions.

erase *[prefix=path] filename*
Erase a specified file from a /var/lib/bayonne prefixed path.

info *[prefix=path] filename [format=¿fmtvar] ...*
This can be used to query and examine a single file. The duration of the file (if audio), date of creation, file type, etc, can all be extracted into seperate variables.

key *%var [index=group] [value=default]*
This is used to map a persistant data key into a local variable. All set operations modify the shared persistant key. The key is saved when the server is shutdown, and reloaded at startup, so it is a global persistant object which is maintained in a save file.

list *prefix=path save=¿results [count=¿files]*
This is used to list the content of a directory and save the results into an array that can be queried.

move *[prefix=path] source destination*
Move or rename an individual file in the /var/lib/bayonne prefixed path.

readpath *%var value ... [prefix=prefix] [extension=ext] [voice=voice]*
writepath *%var value ... [prefix=prefix] [extension=ext]*
This is used to evaluate a path expression for a readable voice file based on Bayonne parsing rules, and set a target variable with a fully qualified filename. The space needed will be created if the var did not exist before, so it is best to not seperately size the given var. These are often used in macro libraries to pre-parse file paths for invoking libexec apps.

17.11 Defined macros

Some commands in Bayonne are actually created through .def files that are loaded by the server. These may appear and can be used as ordinary commands, but in fact are built from macros. They may invoke external programs through libexec, or perform combinational functions.

annotate *file=name text="text" [extension=.ext] [prefix=dir]*
Set the annotation of an audio file. This only applies to .au and .wav files. It is ignored for raw audio files.

audiotrim *file=name [padding=frames] [extension=.ext] [prefix=dir]*
Trim lead and trailing silence from an audio file.

audiostrip *file=name [extension=.ext] [prefix=dir]*
Strip silent frames from audio file, compressing the result.

diskspace *results=&var filesystem=/path*
Check and return disk space for a given file system mount point. This is returned as a simple integer percentage that is saved into a variable through call by reference.

ifconfig *results=&addr iface=ethx [address=newvalue]*
This can be used to get, and optionally set, the ip address of a specified ethernet interface. It invokes ifconfig.

ogg *file=name [prefix=dir]*
Create a .ogg version of the specified audio file in the same directory.

sendmail *[from=who] to=to [cc=cc] text=msg [file=attachment] [prefix=dir] [extension=.ext]*
Deliver a text message, with optional attachment, to a specified user through smtp e-mail. Some parts are controlled by [stmp] configuration in server.conf.

uptime *results=&time*
This can be used to retrieve current system uptime.

url-fetch *from=url file=to [prefix=dir] [extension=.ext]*
Used from url.def. Loads a specified url to a file. This also provides session management, including cookies. A new session is created for each unique telephone call.

url-play *from=url [extension=.type]*
Fetch and play a file from a url. If audio file conversion is needed, it is automatically performed. The file is removed after play completes.

17.12 Other commands

config *key value*
This is used both to define script files that may have keys found in an external config file (/etc/bayonne/scripts.conf), and to set a default value for an individual key if not set from the file.

random.range *[seed=seed] [count=count] [offset=offset] [min=value] [max=-value] [reroll=count] %var ...*
The "random" package offers a fairly complex number of options for creating or storing pseudo-random digits into symbols. These include things that simulate various dice behavior, such as a known sum (count) of a known range of values, and even the ability to specify minimum or maximum values that can be generated.

random.seed *seedvalue*
Seed the pseudo-random number generator. This is used with the "random" package.

replace[.offset] *[offset=bytes] %var find replace ...*
This is used to replace a specific digit pattern with a new value if the digit pattern is found at the specified offset in %var. This is available with the "digits" package.

scale.precision *scale %var...*
The "scale" package enables basic floating point multiplication. This can be used to rescale a known variable by a floating point value, and storing the result to a known digit precision. For example, to scale a variable by 40%, we could use: "scale.2 0.4 %myvar".

sort[.reverse] *[token=char] [size=bytes] %var*
The ccScript "sort" package allows one to sort either packed string arrays, which use a token to seperate content, or the contents of a "sequence", "stack", or "fifo". Sorting can be in forward or reverse order. chop. This is available with the "digits" package.

18 Troubleshooting ccScripts and TGIs

The collect command ***adds*** *to %session.digits, it doesn't overwrite it. Make sure that you're clearing %session.digits before each collect (unless you really do intend to append).*

Don't use '=', use '-eq' to check for equality. Also, '==' is broken in older versions of Bayonne. Use '.eq.' instead.

Are you confusing the name of a script (like "foo") with a label name (like "::foo")?

Remember that the pound sign is used as a comment character. Things like "dial #" don't work because ccscript thinks you're starting a comment. Quote the "#" character instead.

*Make sure you are using the *::foo syntax when playing prompts, and that you have %application.language set properly. "play foo" is almost certainly not going to do what it looks like it should do. Use "play *::foo" instead.*

Make sure that if you use a variable returned by a TGI script that the TGI script defined it. Otherwise bayonne dumps core (as of 0.6.4).

Did the ccscript engine print out any interesting error messages during startup or 'bayctrl compile'? Perhaps you should review them.

Did you remember to run 'bayctrl compile' or to restart bayonne after you modified your script?

If you do things like "goto script", and script.scr looks like this:

```
::start
do stuff
do stuff

^event
^event
goto script
```

The goto will fail. Instead, say "goto script::start".

Make sure that after you deal with an event, the script jumps somewhere. If the path of execution falls off the bottom of the file (or hits another label), then the script engine will jump back to the beginning of the file (or the current label) ad infinitum. Keep in mind that you are developing a telephony application, and you must be constantly interacting with the user or they think you've hung up on them.

When jumping as the result of a conditional (like "if %return -eq 1 goto main"), you don't say "goto". State it in the form "if %return -eq 1 main". The goto is implied after the if conditional.

If you're using Perl and it's DBI module for doing database accesses through TGI, here's one way you can retrieve data from the database via fetchall_arrayref(). The syntax seems to be easily forgettable for some reason.

```
$ref = $sth->fetchall_arrayref();
$row0_col0 = $$ref[0][0];
$row1_col1 = $$ref[1][1];
$row0_col1 = $$ref[0][1];
```

The standard way to get digits so the caller can interrupt the message is:

```
clear %session.digits

play *::1 # "Press 1 for foo, press 2 for baz,
          # press 3 for gronk..."
sleep 60

^dtmf
        collect 1 5 "*#" "ABCD"
        if %session.digits -eq 1 ::label1
        if %session.digits -eq 2 ::label2
        goto ::invalid
```

The standard way to get digits so the caller can't interrupt the message is:

```
clear %session.digits

play *::1 # "Press 1 for foo, press 2 for baz,
          # press 3 for gronk..."
collect 1 5 "*#" "ABCD"
if %session.digits -eq 1 ::label1
if %session.digits -eq 2 ::label2
goto ::invalid
```

** A note on event traps:*

They are order sensitive. If you have

```
^dtmf
        goto ::foo
^pound
        goto ::bar
```

You will never be able to reach bar. `^dtmf` *takes precedence. Also, traps do not work within traps.*

```
^dtmf
        ^star
                goto ::foo
        ^pound
                goto ::bar
```

Will not work. Dtmf detect is always turned off in the script step following a dtmf trap, with the exception of the collect command.

It's a good idea to document your TGI return values in your program header. Make a template for all your TGI programs and stick to it. Make sure there's a section for the return values in the headers and use it. One convention seen around the OST code is to use 1 for a successful call, 0 for an unsuccessful call, and -1 for an internal script error.

Remember that the value of the %return variable is persistent. If you aren't careful, your TGI scripts will fall through without setting a return value. This is especially annoying if you forget to set a return value which means "operation successful" If you don't see a line like this in the server logs:

```
fifo: cmd=SET&2&return&1
```

Then your TGI script isn't setting a return value. The ccscript that's executing your TGI will then use the return value from the last ccscript you executed, which is just hours of debugging fun. Especially when one of your TGIs is working just

fine (but doesn't set a return value) and your ccscript checks the return value to see if an error occurred, and guess what, it's the return value from the TGI script you called before the current one. Chances are that that return value doesn't have anything to do with the return value from the TGI script you just executed, which leads to very confusing results.

Document your database schema. Make sure that you put the column indexes into the database schema document, and you include a Big Fat Warning that tells any potential modifiers of said document that if they touch the document, they get to audit any database access code that uses hard-coded column indexes. The idea is that if they change the database schema, those column indexes may no longer be valid. An even better solution (if your TGI language supports it) is to define a set of symbolic constants for the database columns in one file and include the constant definitions in all the database access code.

19 Phrasebook Rules

19.1 Introduction

Bayonne is provided with a standard "prompt" library which supports prompts for letters and numbers as needed by the "phrasebook" rules based phrase parser. The phrasebook uses named rules based on the current language in effect, as held in "%language" in ccscript.

Phrase rules can be placed in bayonne.conf proper under the appropriate language and in application specific conf files as found in /etc/bayonne.d. English "rules" are found under section [english] in the .conf files, for example.

Phrasebook prompts are used to build prompts that are effected by content. Lets take the example of a phrase like "you have ... message(s) waiting". In english this phrase has several possibilities. Depending on the quantity involved, you may wish to use a singular or plural form of message. You may wish to substitute the word "no" for a zero quantity.

In Bayonne phrasebook, we may define this as follows:

in your script command:

```
speak &msgswaiting %msgcount no msgwaiting msgswaiting
```

We would then define under [english] something like:

```
msgswaiting = youhave &number &zero &singular &plural
```

This assumes you have the following prompt files defined for your application:

- *youhave.au "you have"*
- *no.au "no"*
- *msgwaiting.au "message waiting"*
- *msgswaiting.au "messages waiting"*

The system will apply the remaining rules based on the content of %msgcount. In this sense, phrasebook defined rules act as a kind of "printf" ruleset. You can also apply rules inline, though they become less generic for multilingual systems. The assumption is that the base rules can be placed in the [...] language area, and that often the same voice prompts can be used for different effect under different target languages.

The speaking of numbers itself is held in the default Bayonne distribution, though the default prompt list can also be replaced with your own. Rules can also appear "within" your statement, though this generally makes them non-flexible for different languages.

Speaking of currency "values" have specific phrasebook rules. Currency values are also subject to the "&zero" rule, so for example:

balance=youhave &cy &zero remaining

and using:

play &balance %balance nomoney

can use the alternate "no monay" .au prompt rather than saying "0 dollars".

19.2 English

The following default phrasebook rules are or will be defined for english:

&number	*speak a number unless zero*
&unit	*speak a number as units; zero spoken*
&order	*speak a "order", as in 1st, 2nd, 3rd, etc.*
&skip	*skip next word if spoken number was zero.*
&ignore	*always ignore the next word (needed to match multilingual).*
&use	*always use the next word (needed to match multilingual).*
&spell	*spell the word or speak individual digits of a number.*
&zero	*substitute a word if value is zero else skip.*
&single	*substitute word if last number spoken was 1.*
&plural	*substitute word if last number spoken was not 1.*
&date	*speak a date.*
&day	*speak only day of month of a date.*
&daydate	*speak day of week and day of month.*
&fulldate	*speak day of week and full date.*
&weekday	*speak the current day of the week.*
&time	*speak a time.*
&hour	*speak abbreviated time, just hour.*
&duration	*speak hours, minutes, and seconds, for duration values.*

19.2.1 Number Prompts

0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 30, 40, 50, 60, 70, 80, 90, hundred, thousand, million, billion, point

19.2.2 Order Prompts

1st, 2nd, 3rd, 4th, 5th, 6th, 7th, 8th, 9th, 10th, 11th, 12th, 13th, 14th, 15th, 16th, 17th, 18th, 19th, 20th, 30th, 40th, 50th, 60th, 70th, 80th, 90th

19.2.3 Date/Time Prompts

sunday, monday, tuesday, wednesday, thursday, friday, saturday
january, february, march, april, may, june, july, august, September, october, november, december
am, pm

19.2.4 Currency Prompts

dollar, dollars, cent, cents, and, or

20 Copyright

Copyright (c) 2005 David Sugar.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts