GNU Bayonne Script Programming Guide

David Sugar
Open Source Telecom.
sugar@gnu.org, http://www.gnu.org/software/bayonne

2003-01-03

Contents

1 Introduction

GNU Bayonne is a script driven telephony application server. As such, it has it's own scripting language built by class extension from the GNU ccScript interpreter. 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, GNU ccScript 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 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), we looked for 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. Most script interpreters, besides being non-deterministic in their behavior, also do not support direct single stepping in that manner.

In GNU ccScript, 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.

Finally, we 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. GNU ccScript forms a single interpreter image by compiling all scripts at once directly into memory. Furthermore, since we did not wish to have the server "down" to load a new script image, we 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 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 statement, 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. Starting with ccScript 2.4.4, it is not limited in any manner whatsoever. 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, 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.

3 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 can be in the form "::label" or "label:".

A script file that has commands which need to branch to a label can do so by branching to the local script file name. Hence, if there is a "test:" entry in "your.scr", another script command can branch to the test: entry directly by using ::test, as in "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 "test:", I can branch to it

by refering to myscr::test. Hence, I could use “goto myscr::test” for example.

Script labels can be made private within a given script file using the “.private” directive. This means that they can only be referenced from other scripts within the same file in the “::xxx” form, and not externally through “yyy::xxx” references. By default all script labels are public. You can also use “.public” as a directive to convert the next script labels in a file to public after using “.private”.

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 with the “.export” directive. Exported scripts are named, compiled, behave, and are referenced as if they appeared in the file named in the .export statement rather than the .scr file they appear in.

4 Symbols

Bayonne scripting recognizes three kinds of symbols; constant ”values” (literals), %variables, and @indirection. In addition, Bayonne recognizes compile time substitutions, known as $names, which can substitute to any of the above three. 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, while others may be freely created by scripts themselves. However, all variables and values stored, with the exception of globals, 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.

Bayonne also recognizes symbol scope. Local symbols are created within symbol scope, resulting in unique instances within subroutines. In addition, all local symbols in symbol scope are removed when returning from a subroutine. global 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.

A fourth class of symbol exists only at compile time, $defined symbols are substituted when the script file is compiled, and usually reduce to a simple constant, though variables can be named for the compiler. All constants are defined in the [script] section of bayonne.conf.

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 %foo.xxx and %foo.yyy. When in another script file, one can reference the full name, %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
%script.trigger	name of armed symbol that was triggered
%return	process exit value of last libexeced TGI script
%session.id	global call identifier
%session.date	current date
%session.time	current time
%session.digits	currently collected dtmf digits in the digit buffer
%session.count	count of number of digits collected
%session.starttime	time of call starting
%session.startdate	date of call starting
%session.duration	current call duration

%session.callerid	generic identity of calling party
%session.calledid	generic identity of how they called us
%session.home	initial script invoked
%session.schedule	script that was scheduled
%session.eventsenderid	port that sent a message with the “send” command
%session.eventsendermsg	the message given to the “send” command
%session.language	current language in effect (default = ”english”)
%session.voice	current voice library in effect (default = ”UsEngM”)
%session.joinid	last port we successfully joined with.
%session.parent	id of call session that started current one.
%session.pickupid	last port that picked us up.
%session.transferid	port that is being transferred to us.
%session.recallid	port we successfully picked up.
%session.extension	extension number call is part of.
%session.loginid	effective user id for preferences database
%user.*	preference database entries for this user
%line.*	line settings we may have in effect
%driver.id	driver timeslot associated with call session.
%driver.card	physical card number associated with session.
%driver.span	T1/E1 span number associated with session.
%driver.network	Type of port switching; ”none”, ”soft”, or ”tdm”
%pstn.dnid	did/dnis number dialed if available
%pstn.clid	ani or caller id number if available
%pstn.name	caller name if passed in callerid
%pstn.redirect	if origin is a telco redirect
%pstn.ringid	if distinctive ringing is available
%pstn.infodigits	if telco infodigits were passed to us
%pstn.rings	number of rings so far
%pstn.interface	Type of pstn interface; analog, digital, etc.
%pstn.tone	Last special pstn tone detected.
%policy.name	name of policy for this session
%policy.member	nth member of this policy
%policy.*	various policy variables.
%audio.volume	volume level in 0-100 for play and record.
%audio.extension	default audio file extension (.au, .wav, etc)
%audio.format	default audio format (ulaw, alaw, g721, etc)
%audio.annotation	annotation of last played or for recorded file
%audio.played	samples played from played file

%audio.recorded	samples recorded to record file
%audio.created	date played file was created
%audio.timeout	timeout in a play wait
%audio.trim	minimal number of samples to trim at end of file
%audio.offset	sample offset of play or record at end of command
%audio.buffer	driver specific buffering for computing offsets
%server.ports	total ports on bayonne server
%server.version	version of bayonne server
%server.software	software identifier; "bayonne"
%server.driver	which driver we are running
%server.node	node id of our server
%rpc.status	last posted rpc status result
%rpc.loginid	rpc requestor's login identifier
%rpc.expires	time period for rpc to complete by
%sql.driver	name of loaded sql driver
%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

A special set of variables may be created by the application program under the name %global. global.xxx 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.

4.1 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.

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.

In addition to type behavior, symbols have properties that may be associated with them. These include a number of default properties, and some that may be available through the loading of ccscript packages. A symbol’s property may be extracted by adding a .property to the symbol name. One such property is the symbol ”type”. Hence, to access the symbol type of %myvar in a script command argument, one might use “%myvar.type”. The following script properties are defined:

property	requires	description
.type	–	base symbol type in scripting
.length	–	length of symbol contents
.size	–	size of symbol entry
.count	–	number of members in array or stack
.max	–	maximum size of array or stack
.value	–	computed integer value of symbol contents
.bool	–	compute bool value of contents
.dtmf	(bayonne)	returns dtmf representation of contents
.upper	.use string	returns upper case text of contents
.lower	.use string	returns lower case text of contents
.capitalize	.use string	returns capitalized text of contents
.trim	.use string	returns contents without lead or trailing spaces
.url	.use url	decode url escaped content
.bin	.use url	decode binhex content
.each	.use digits	extracts content into a , seperated list of chars
.date	.use date	returns date encoded values from content
.year	.use date	returns integer year of a date variable
.month	.use date	returns integer month of a date variable
.weekday	.use date	returns weekday of a date variable
.monthof	.use date	returns named month of a date
.day	.use date	returns integer day of month of a date
.time	.use time	returns time encoded values from content

.hour	.use time	returns hour value from time
.minute	.use time	returns minute value from time
.second	.use time	returns second value from time

Similarly, some properties may be used to set variables of a specific type or to convert the values stored in a variable through a set command into a specific format. These are done by using "set.xxx" where xxx is a specific property format. For example, to url encode a string, one can use "set.url %mystr ...contents...". A date variable to store the current date may be similarly created with "set.date %myvar".

4.2 Arrays, lists, and Hashes

While not meant for intensive algorithmic work, Bayonne's scripting does support a concept for both handling arrays and lists. An array is considered a numerically sequenced list of variables with access controlled by a common .index property. A list is a token packed list of character strings.

There are also other ways of creating array "effects". One way to do this is to use symbol indirection. Symbol indirection means the symbol that is referenced is found by examining the contents of another symbol. For example "sim 1" string will be logged after execution of this fragment:

```
set %myref "mysim1"
set %mysim1 "sim 1"
slog @myref
```

When using lists, Bayonne assumes by default that these are comma seperated content in a symbol. However, characters other than commas may be used. A list may be examined in a variety of ways. Using the "string" package, it is possible to build, extract, and organize the contents of lists. The "foreach" keyword can be used to look through the contents of a list. In each case, a special token= attribute may be used to specify the seperator character being used if it is not a comma.

The default list seperator token, ",", is stored in a special variable, %script.token. You may modify this single character variable to another token. The special .n properties can be used to access a part of a token seperated list in a manner similar to how members of an array are accessed. List members are numbered from 1, so xxx.4 will access 4-th member of xxx list.

Starting with ccScript 2.4.4, it is also possible to compute a variable hash as part of a symbol name reference. For example, a variable named "%var#hash" will be expanded to use the contents of %hash to compute the name as a componentized name. Hash tables in this form may appear in either local or global storage, and hence provide a means to componentize local scope symbol names and store stack local arrays.

4.3 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.parent 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.

5 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 \verb`ŝtar=	–	dtmf "#" or "*" key hit
`^tone`	–	tone event heard on line
`^signal`	detach	notify while waiting for other trunk
`^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
`^answer`	–	call progress caller answered
`^busy`	–	call progress dialed line busy
`^noanswer`	–	call progress call timed out
`^time`	–	call timer event trap

Some of these script events also have Bayonne ccScript 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.

6 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:

conditiona	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
-altvoice	tests if a given voice exists in the alt prompt directory
-sysvoice	tests for a given system voice library
-appvoice	tests for a given application voice library
-group	tests if a specified trunk group exists
-plugin	tests if a specified plugin is loaded
-service	tests if the service level is set to a specific value
-dtmf	tests if a specific dtmf "option" setting is in effect
-feature	tests if the feature specified in argument exists
-ext	whether the argument refers to a valid extension number
-station	whether refers to a station port extension number
-virtual	whether refers to a virtual extension entity
-user	whether the argument refers to a user profile id
-dnd	whether dnd is set for the extension argument
-hunt	whether the argument refers to a hunt group

The -feature option can test for a number of features. The features you can test for include "-feature join" to test for join support, "switch" for pbx support, "spans" for digital span support, and various audio capabilities.

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”, “foreach-loop” and, with ccscript 2.5.2 and later, “fordata-loop”. The for statement assigns a variable from a list of arguments, much like how for works in bash. foreach can be used to decompose a token seperated list variable. fordata is used to perform a “read” statement from data statements (such as a #sql query result) directly in a loop. In all cases, break and continue can still be used within the loop.

7 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 invoked through either the “gosub” or “call” script statements. When a “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 through “call”, it is possible to specify if a new local variable heap will be created. If a new heap is created, local variables are 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’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 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 ::mysub myvar=3", for example. When this is done, a local constant is created, known as %myvar, that is then usable from ::mysub, and exists until ::mysub returns. Since this is a constant, its value may not be altered within ::mysub.

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:

```
::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 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 ::mysub myref=&mysym
        slog %mysym
...

::mysub
        set %myref "tested"
        return
```

In this case, the slog will show "tested" since %myref in ::mysub actually points back to %mysym in the calling script.

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 either using the "source" statement, or "call.local". It is also possible to have a subroutine that has no local variable context, and hence always refers to the base global context. This is done with a "call.global".

8 Transaction Blocks

In addition to 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.

9 Files, paths, prompts, extensions, and directories

In the default configuration supplied with 1.2.0, there are three places that audio prompts may be played from; a subdirectory of /var/lib/bayonne, a subdirectory of /usr[/local]/share/bayonne, and a subdirectory of /home/bayonne. /var/lib/-bayonne is meant to hold audio that is manipulatable and recorded samples. /usr/share/bayonne includes both system and language specific prompts that are supplied by pre-installed or packaged Bayonne applications. /home/bayonne is meant to store both scripts and audio prompts that are locally created by the

system administrator for site specific or custom applications.

To refer to an audio sample stored in a /var/lib/bayonne/xxx 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 preprocessor directive, **.prefix**, or **.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. However, you can either modify %audio.extension in a running script, and many commands allow you to override the default extension. 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 lot of special magic occurs to determine where the audio file will be played from. The first consideration is the current voice setting. Since audio samples are split into language specific voice libraries, each with a seperate subdirectory in /usr/-share/bayonne, the directory looked at is the one that the current voice library is set for. The default is usually UsEngM, and this is set in the [script] section of bayonne.conf as well.

In addition to voice libraries, each application may have a seperate subdirectory for it's own prompts under each voice. The application subdirectory is the same as the base name of the script file (or the effective name if it has been .exported). Hence, if we have an application, "myapp.scr", and we try to use "play myprompt", then, the first place examined would likely be /usr/share/bayonne/-

UsEngM/myapp/myprompt.au. If there is no application specific directory, then the main voice library directory is used. Hence, if we use "play 1", and there is no /usr/share/bayonne/UsEngM/myapp subdirectory, then instead we will retrieve and play /usr/share/bayonne/UsEngM/1.au.

The current voice libary may be found in %session.voice. While %session.voice may be changed directly, the "options voice=xxx" script command is usually used to select a new voice because it will also select the correct phrasebook language module to use with that voice. The default extension of voice library prompts is controlled by the driver, and is usually either .au, for sun audio encoded mu-law files, or .al, for raw a-law encoded copies of .au content. Furture drivers may also make use of .gsm or .adpcm for pre-encoded audio prompts.

Audio may be played directly from a URL when XML support has been enabled in Bayonne. URL audio is played at the moment only from a http: source. hence, one can do something like "play http:/audio/myaudio.au". If XML support is not enabled, then 'http:' prompts are ignored.

In addition to standard URL's, there are a number of special pseudo-urls's that have special meanings. These are in the form of xxx:yyy, where xxx is a special url code, and yyy is an autoprompt, or pathname.

The following special url codes are defined:

alt:
This redirects prompt selection to /home/bayonne. From that point, the actual prompt file is then selected based on the same voice library and script path selection rules as if it had been taken from a standard prompt stored through the default path. Hence, a subdirectory of /home/bayonne named after the current voice library will be used.

app:
This refers to selecting audio from a subdirectory named the same as the application script name from a voice library subdirectory from /usr[/local]/share/-bayonne.

audio:
Somewhat related to **music:**, the **audio:** name refers to any .au or .al file that has been stored in /usr/[local]/share/bayonne/audio. These audio samples can be any generic audio file that may be desired, including sound effects, etc.

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.

music:
This references audio samples stored as digital music. Digital music is typically used for playing of "music on hold". Digital music is stored in /var[/local]/share/-bayonne/music. It is stored in 8 bit encoded u-law (and optionally in a-law) since 8bit 8khz files are actually smaller than mpeg encoded high fidelity audio, and because the telephone network does not support higher quality audio in the first place.

sys:
This refers to audio prompts stored in the non-language specific "system" voice library. These are found in /usr[/local]/share/bayonne/sys.

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

usr:
home:
This refers to audio stored in the alternate audio prefix (altprompts in bayonne.conf). The alternate audio prefix is normally set to /home/bayonne. The actual file will either be loaded from an application specific subdirectory of /home/-bayonne if a subdirectory exists to match the script filename, or directly from the /home/bayonne directory itself.

var:
This refers to a subdirectory of /var/lib/bayonne that is named the same as the current script file. This provides a quick way for a script to organize it's recordable prompts under the same subdirectory of the datapath.

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 system prompt directory. This is often used to organize system audio prompts used in system support scripts, such as playrec. The actual location is found from /usr[/local]/bayonne/sys/xxxx/.

xxxx::
There are a number of other odd forms of pathnames used to select prompts based on script names or so called export names. This translates roughly to a language localized prompt based on the voice library name that is found in /usr-[/local]/share/bayonne. This allows the xxxx name to be used in place of the

name of the current script file.

/
When bayonne is used to host user driven scripts, this is done in a manner somewhat anagelous to how apache hosts the / path and user specific content. The audio prompt for these are assumed to exist in a subdirectory of a user's home account, usually from $HOME/.bayonne.

10 Pre-processor directives

Based on ccscript 2.5.0 and later, there are two types of pre-processor directives; those that are compile control directives, and those that involve compile-time symbol or token substitution. Compile control directives start with "." and are on a seperate line, along with any options or arguments, such as filenames. An example of a compile control directive is ".include xxx", which can be used to insert include files.

Compile time symbols are given a value at the time the script is compiled, and may be considered like a constant. Some compile time symbols are constants, and may be used to represent values found in the [script] section of bayonne.conf. For example, to insert the definition of "timeout" that is found in [script], one might refer to it within ccScript as "$timeout".

With ccScript 2.5.0, we introduced three special compile time symbols to help with debugging. These are "$script.name", "$script.file", and "$script.line". The first refers to the name code is currently being compiled under. With the use of the ".export" directive, this may in fact be different than the filename, which may be found with "$script.file". The final symbol expands to the current statement line number being compiled.

The following pre-processor directives exist:

.code
.data
.map
Define the next section of text in your source file either as script (code), as a local embedded data block for use with a "read" statement, or as a special embedded data block for use with the "map" statement.

.default[nn] *name [default-value]*

This creates a definition for a user preference entry. User preference entries are stored under /var/lib/bayonne/users/xxx, and contain persistent user information such as account passwords or telephone extension settings. The default value, if present, is used to specify the default to fill a new user preference record for this named entry when it is created. These may be accessed through the %user. global depending on the login state of a given call session.

.define *name value*
Used to define the value of a pre-processor symbol that can then be referenced under $name.

.export *name*
This is used to export the remaining ::xxx script script sections as if they were compiled from a script source file that is named by the directive rather than the one they are compiled from. Hence, the sections are compiled as name::xxx rather than the source file name that they are compiled from. This can be used to override or insert a section script into another script file.

.include *filename ...*
Insert text from another source file. This can be used to insert constants such as **.define** statements from a seperate header file, for example. The include directory is assumed to be the same as the script source directory, although one can specify relative paths for subdirectories.

.hunt[nn] *name [default-value]*
This is used to create persistent hunt group entries and definitions that are used for pbx hunt groups, and that are saved under /var/lib/bayonne/hunting/xxx. These are usually extrated through the huntinfo command in scripting.

.initial[nn] *name value ...*
This is used to define script global symbols that need to be of a known size and initialized to a known value at the start of each call session before scripts are ran. If you have a **.initial** statement in test.scr for a variable named *save*, for example, a global variable named %test.save will be initialized for each call.

.languages *language ...*
Load Bayonne phrasebook module for a specified language if it is not already installed because the current script depends on it.

.line[nn] *name [default-value]*
This is used to create an entry for port (line) persistant entries that may be used to save line properties that are modifyable. These are typically accessed through the %line. variables.

.module *modulename ...*
Load the specified Bayonne "module" (.mod) if it is not already installed in memory because the given script depends on it being active.

.prefix *subdir ...*
.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.

.private
Define any ::xxx scripts compiled from this point forward as being private and only referencable from within the current script file. This means one cannot use yyy::xxx to invoke or reference a local script. The **.public** directive can be used to re-enable public access to compiled scripts in the current file. This directive only effects the current file, and is reset to public when the next source file is compiled.

.public
Mark any scripts compiled from this point forward as being public. This is used to re-enable yyy::xxx scripts if a **.private** directive has been used.

.requires *commandname*
Tests to see if a specified and required script command exists in the current system. If the required script is not found, then the current file cannot be compiled, and the system halts.

.rpc *rpcname*
Bind the next script statement as a RPC named script service to be invoked through Bayonne rpc services under the specified name. This assumes the script returns a status message back to the rpc server.

.rule *rulename rule*
.english *rulename rule*
.lang... *rulename rule*
Define a phrasebook rule in your sourcefile. This allows in-source definition of rules that your script may use in the "speak" command, rather than having to define them seperately in a config file. Rules that apply to all languages are defined with a single bf .rule directive, while language specific varients, if needed, can also be specified by using the language name as a "." preprocessor directive.

.safe
.unsafe
This is used to mark sections of code whether they should report compile time

syntax errors or not. Unsafe code will report no error, and may be used when loosly binding optional script keywords that work only when specific modules are present.

.template *templatefile*
This allows a foreign script file to specify the event handlers that will be defined and used for the current script. This allows one to define a template script for common event handling shared among a number of common scripts.

.use *packagename ...*
Install a generic ccscript package module that the current script requires, if it is not already installed. These are normally found in /usr/lib/ccscript2.

.xml *xml-modulename ...*
Load the specified Bayonne XML support module (,xml) if it is not already installed.

11 Command Reference

11.1 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.

alias *%name value*
Create an alias symbol name that points to a real symbol name when referenced. The alias and target must be in the same scope, hence local aliases cannot reference global objects in subroutines.

array[**.size**] *count [values...]*
array *size=size count=count %var [value ...]*
Create a ccScript array composite object. This creates a special %var.index to select a specific array element, and a bunch of seperate variables named %var.1 through %var.count, one for each array element.

cache[**.size**] *count %var [value ...]*
cache *size=size count=count %var [value ...]* Create a ccScript fifo cache variable. This creates a variable that logs and returns content in reverse order.

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.

const *%var values...*
const *var=value ...*
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.

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

init *%var values...*
init *var=value ...*
init[**.min—.max**] *[size=bytes] %var [=] values...*
init[**.size**] *%var [=] values...*
init[**.property**] *%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. Can also init a variable to a minimum or maximum value of a set of values. A property plugin may also be initialized, such as initing a ".date" or other specialized plugin type value. The "xx = expr" form can be used to initiate a numeric expression with basic +, -, *, and / operators.

lifo[**.size**] *count %var [value ...]*
lifo *size=size count=count %var [value ...]*
stack[**.size**] *count %var [value ...]*
stack *size=size count=count %var [value ...]*
Create a ccScript "stack" variable. Stack variables are lifo objects and unwind automatically on reference.

list[**.size**] *[size=bytes] [token=char] %var values...*
List is used to quickly construct a token seperated packed list variable which may then be used with the foreach loop, referenced through the packed property index, or manipulated with the string command.

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 by building a target public object name from components that are then glued with "." notation. This is different from an alias since a simple named alias can only reference the local scope in a subroutine.

sequence[**.size**] *count %var [value ...]*
sequence *size=size count=count %var [value ...]*
Create a ccScript repeating sequence variable. This repeats content of a sequence in the order set.

set *%var values...*
set *var=value ...*
set[**.min**—**.max**] *[size=bytes] [justify=format] %var [=] values...*
set[**.size**] *[justify=format] %var [=] values...*
set[**.property**] *%var values*
Set a variable to a known value. Can also set multiple variables. Can also set a variable to the minimum or maximum value of a set of values. A property plugin may also be directly set, such as initing a ".date" or other specialized plugin type value. The "xx = expr" form can be used to set a symbol based on a simple numeric expression using basic +, -, *, and / operators. A left, right, or "center" justification may also be used. Please note, in the future, the overloaded expression form may be seperated into a new "number" and "float" keyword.

size *space %var...*
Pre-allocate "space" bytes for the following variables.

11.2 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.

arm *%var*
Arm a variable so that it will auto-branch if modified.

dec[**.property**] *%timevar [offset]*
Decrement a variable, perhaps with a specified offset, otherwise by "1". When used with packages that set numeric properties, such as the "date" and "time" package, dec can adjust dates and times as well.

disarm *%var*
Disarm an armed variable.

dup *%var %dest*
Duplicate an existing object into a new one.

inc[.property] *%timevar [offset]*
Increment a variable, perhaps with a specified offset, otherwise by "1". When used with packages that set numeric properties, such as the "date" and "time" package, inc can adjust dates and times as well.

post *%var value ...*
Post one or more additional values into a stack, fifo, sequence, or cache.

swap *%var %var*
Exchange the contents of two variables with each other. They must both be of the same scope.

remove *%var value ...*
Remove a specific value entry from a stack, fifo, sequence, or cache variable.

dirname *%var*
Much like the shell dirname, to extract a directory name from a path.

basename *%var [extensions...]*
Reduce a variable to a simple base filename, and strip any of the optionally listed extensions from it.

fullpath *%var fullpath*
If %var is a partial pathname, then merge it with the passed path to form a complete pathname.

11.3 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:

begin

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

disable *name=label trapids...*
This is used to disable specific :.. handlers within a live script. The script, when used, behaves as if there never was a :.. handler programmed for it. The **enable** command can be used to make the handlers active again.

enable *name=label trapids...*
This is used to enable specific trap handlers within the body of a script which may have been disabled.

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.

lock[.wait—.unlock] *id=value*
Attempt to create a global lock under the specified id. If successful, then no other active call will be able to claim the same global lock until the current call session releases it, either through a **lock.unlock** or by exiting. If .wait is used, this command will block until the lock is aquired. Global locks could be used for pin numbers, account ids, etc.

11.4 Looping, Branching, and Conditionals

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

call[.public—.protected] *value [var=value ...]*
gosub[.public—.protected] *value [var=value ..]*
Call a named script or event handler as a subroutine. If the call is successful, an optional list of variables can be conditionally set. ".public" can be used to specify that the subroutine is ran without a local stack, and ".protected" can be used to share the current stack with the subroutine directly rather than having it create a new local context. Passed arguments are set in the local context of a new subroutine as constants. It is also possible to pass variables by reference to a subroutine.

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
Terminate script interpreter or invoke the special "::exit" label in the current script.

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".

fordata *source—table=source %var ...*
Read "data" statements starting from a script referenced as the "source" as part of a loop. The loop will exit when all data lines have been read. This command could be used to create a simple loop to read #sql query results, for example. "fordata" was introduced in ccscript 2.5.2.

foreach[.offset] *%var %packed*
Assign %var from items found in a packed list. An optional offset can be used to skip the first n items. A foreach loop may include **case** statements.

goto *label [var=value ...]*
Goto a named script or event handler in a script. If the goto is successful, an optional list of variables can be conditionally set.

if *value op value [and — or ...] label [var=value ...]*
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. Optionally a set of variables can be initialized based on the conditional branch. 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.

label *value ...*
Create a named local label to receive a skip command request.

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

on *trap label*
Allows one to test for a previously blocked signal and see if it had occurred. If the signal had occurred, then the branch will be taken.

once label
Within a single script, once is guaranteed to only goto a new script (like a goto) "once", which can be used as protection agains recursive invokation.

pop
Pop a gosub off the stack. This only has an effect when returning.

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

return[.exit] *[label] variable=value ...*
Return from a gosub. You can also return to a specific label or ĥandler 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.

select *value match label [..match-label pairs] [var=val...]*

Select is used to branch to a new script based on matching a value or variable to a list of possible values. For each match, a different script or event handler may be selected. Options include "select.prefix" to match string prefixes, "select.suffix" to match by tailing values only, "select.length" to match by length, and "select.value" to match by numeric value. If a branch is found, an optional list of variables can be conditionally set.

skip *[label]*
Skip may be used alone to skip over a case statement. This is similar to what c/c++ does with cases if no intervening break is present. Since ccScript automatically behaves as if break is present, skip is needed to flow into another case. The second use of skip is to branch to a local label as defined by the **label** keyword.

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.

try *lables... [var=value ...]*
Attempt to goto one or more labels. If the label is not found in an existing script, the next one in the list will be tried. If any branch attempt is successful, then optionally variables may also be set.

tryeach[.offset] *%packed [var=value ...]*
Attempt to branch to a script based on the values held in a packed list. Each item will be tried in term, starting from the offset if one is specified. If a branch point exists an optional list of variables may be conditionally initialized.

11.5 Basic data sets and logging

GNU ccScript supports some basic manipulation of embedded data tables within a script. This has been extended in Bayonne to support retrieval of multi-row queries that can then be examined in scripting.

data *values...*
data.mapindex *label [variable=value] ...*
This can be used to embed compile time data tables into a script file. Some modules (such as sql query) generate data dynamically for the interpreter and then embed the results also as data statements. the data statements can then either be mapped or read by referencing the script label they appear under. Each data statement line is treated as a seperate row.

map[.prefix—.suffix—.value—.absolute] *[table=label] value*
Map allows branching to a data.mapindex entry in the specified label, or within the current script label if none is specified. A match, either by value, or from start or end, may be made of the given value, and if a matching data.mapindex entry is found, then the label specified is branched to, with the optional initialized variables, as if a goto had been used.

read *[table=source] [row=row] [col=offset] %var ...*
Read a row (line) of data from a data statement, either from the next row of the current table source, or from a specified table source or row offset. The row is read into variables, one for each column. A column offset may be used to start from a specified column offset. Data sources are always referenced by a script label. The special "#" script label (and "#xxx" entries) refer to Bayonne's dynamic script buffer when XML support is enabled, and where query results of various types may be stored.

slog[.info—.err—.crit—.debug *message...*
Post a message to the system log as a SYSLOG message. The logging level can be specified as part of the command. 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.

11.6 Package based extensions

A number of interesting script commands are available through the use of external ccscript "packages". These are typically saved in /usr/lib/ccscript2, and are installed into the interpreter with the .use command. The following extension based commands can be available:

chop[.offset] *[offset=bytes] %var bytes*
This is used to chop out a specific count of digits from within a string. This is available with the "digits" package.

delete[.offset] *[offset=bytes] %var values...*
Delete specific digit values from the specified variable if they exist at the offset specified. For example, to remove a lead "1800" from a phone number, one might use: "delete.0 %phone 1800". If the specified pattern is not found, it is not removed. The offset can be a number, or "end", to specify removal of values from the end of the %var involved. This is available with the "digits" package.

insert[.offset] *[offset=bytes] %var value...*
Insert digit values starting at a known offset in an existing variable. This is available with the "digits" package.

prefix[.offset] *[offset=bytes] %var value...*
If the specified prefix value does not exist at the specified offset, then it is inserted. This is available with the "digits" package.

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.

string.cut—.chop *[token=char] [offset=items] %string %var ...*
This is functionally similar to **string.unpack** except that as each item is unpacked into a target variable, it is also removed from the original packed list.

string.pack—.clear *[token=char] [offset=item] %string [size=bytes] [prefix=text] [suffix=text] [mask=format] [fill=text] values...*
Create a packed string from a list of items, using the specified token to seperate each item (or the default token, ','). If the string already exists, you can pack

items from a specified offset. If the variable doesn't exist, the size can be used to specify the size of the new variable. Prefix and suffix allow each item to have a prefix or suffix in addition to the token. An existing variable may also be appended to or cleared. A special mask may also be used to preformat data in a specific way, such as to filter and field numeric values, zero fill, etc. A fill string can be specified to fill to the remaining end of the object. This keyword is supported with the "string" package.

string.unpack *[token=char] [offset=item] %string %var...*
This can be used to unpack a tokenized string into seperate variables. One may specify the item number to start from rather than the start of the list. This is available with the "string" package.

trim[.offset] *[offset=bytes] %var bytes*
This is used to trim leading and trailing digits outside of the range of count and offset specified. It's the inverse of chop. This is available with the "digits" package.

11.7 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.

answer *[rings [ringtime]] [fax=label] [station=id]*
answer *[maxRing=rings] [maxTime=ringtime]*
Answer the line if it has not been answered yet. Optionally wait for a specified number of rings before answering, and allow a maximum inter-ring timeout before considering that ringing has "stopped". If the rings are stopped before the count has been reached, then the call is treated as a hangup (abandon). If a fax tone is detected, it is possible to branch to an alternate label, as well as to present a fax station id, assuming the driver supports fax.

audit[.log—.clear—.post] *key1=value1 key2=value2 ...*
Declare or post a CDR record. When a cdr record is declared, it is written when the call terminates using the audit plugin. When posted, it is written immediately through the audit plugin to a seperate auditing data spool. If a cdr record had been set earlier, it may be cleared with .clear, in which case no cdr will be posted when the call session terminates.

busy.port—.group—.span—.card *id=ports*

This is an admin priviledged command that is used to busy out a series of ports other than the current one. The effect of busy can be countered with applying the idle command to the same ports.

cleardigits[.count] *[label [var=value ...]]*
Clear a specified number of digits from the dtmf input buffer, and then optionally branch to a specified label as if a **goto** statement has been executed. In place of count, one can use ".all", to clear all digits from the input buffer or ".last" to clear just the last input digit. Furthermore, ".pop" may be used to pop off the first input digit only, and ".trap" may be used to perform dtmf trap handlers after branching or within the current script.

collect[.clear—.trim] *digits [timeout [term [ignore]]]*
collect[.clear—.trim] *[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.

control *command [arguments]*
A priviledged command which allows a script to directly insert any valid fifo control command to the Bayonne server.

debug message..
Send a message to the debugging monitor if one is installed through the plugins.

dial *[timeout=cptimeout] [origin=telnbr] [prefix=prefixcode] 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.

dial.dtmf—.pulse—.mf *prefix=digits] number...*
This is used to perform "soft" dialing operations, which are used to emit dtmf digits and special dialing characters directly as synthesised audio with timed pauses, and to do so without any call progress detection.

dial.int—.nat—.loc *[timeout=cptimeout] [origin=telnbr] [prefix=prefixcode] - number...*

This makes use of the trunk group definitions of special prefixes and codes for international, national, and local dialing, to produce a final valid phone number. The phone number is then dialed through the network, and call progress is used to determine the results.

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

examine[.ext—.trk—.tie] *id=callid %var*
This is used to copy the contents of variables from another call session into the local call session. The use of .ext, .trk, or .tie occur only when used with a PBX capable driver.

flash offtime ontime
flash *offhook=offtimer onhook=ontimer*
Flash the line a specified number of milliseconds (offtime) and then wait for ontime. If dialtone occurs, then may be used as a branch.

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

hangup.port—.span—.card—.group *id=ports*
This is used to hangup on an active call on another port. A specific port or a specific group of ports may be specified.

idle.port—.group—.span—.card *id=ports*
This is an admin priviledged command that is used to reset to idle a series of ports other than the current one.

libexec[.once—.play] *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.

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

options *option=value ...*
The options command is an odd command. It can include common options that

affect script processing, and also options that may be driver specific. An option is set by option name, with a specified value. The common generic options include dtmf, which can specify dtmf handling, either as on, off, line (default), or script global. Another important option is result=, as this is used to send a result back from a rpc initiated script to a web service. The logging= option sets the default logging level. The options keyword can also set a new voice, and when voice= is used, the correct phrasebook language module is also activated.

[alt]play[.any—.all—.one—.tmp] *[prefix=path] [offset=samples] [limit=samples] [gain=db] [pitch=freq-adjust] [speed=slow—fast] [volume=%vol] [text=message] [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 **altplay** *version of this command is used in conjunction with* **say**, *and plays only if there is no tts system installed.*

record[.append] *[prefix=path] [gain=db] [volume=%vol] [trim=samples] [minSize=samples] [text=message] [maxTime=maxrectime] [exit=termdigits] [encoding=audio-format] [annotation=text] [extension=fileext] [save=savename] [offset=samples] audiofile*
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.

record *[prefix=path] [maxTime=timelimit] frames=count filename*
This version of record is used to create a looping audio feed that can be used to share an audio source with multiple listeners on different call sessions.

redirect[.digitcount] *label*
This is a special marker token. When a ĥandler is followed immediately by a redirect statement, the specified number of digits are cleared from the input buffer, and the script branches immediately to the specified label, rather than step executing. This allows for an immediate branch execution, rather than the extra step delay required when a goto immediately follows a ĥandler. The redirect command also behaves as a cleardigits command.

say *[gain=db] [volume=%level] [voice=ttsname] text...*
If there is a tts module installed in Bayonne, or an external one has been made

active, then the say command may be used to generate synthesised speech. This command is ignored if there is no tts service present, and so may be used in conjunction with **altplay** *and* **altspeak**.

send.copy *id—gid—ext—trk—tie=id %var...*
This is used to copy the contents of current variables into the global variable space of another selected call session.

send.digits *id—gid—ext—trk—tie=id digits...*
This is used to send digits into the %session.digits input buffer of another call session, to thereby act as if dtmf digits were detected.

send[.message] *id—gid—ext—trk—tie=id message=text*
This is used to post a message from the current call session to either a specific call session by id, or to a series of active call sessions under a common trunk group. The message recipient branches to a êvent handler, receives %session.eventsenderid with the id of the sending call, and %session.eventsendermsg with the message text.

send.post *id—gid—ext—trk—tie=id %varname value ...*
This is used to post values into the contents of variables in another call session.

service *level*
This is an admin priviledged command which is used to set the service level of the server as a whole, either up, down, or under a special service condition.

sleep *timeout [rings]*
sleep *[maxtime=timeout] [maxRing=rings]*
Sleep a specified number of seconds. Can also sleep until the specified number of rings have occurred. Millisecond timeouts may also be specified using ms, as in "100ms".

[alt]speak[.any—.all] *[language=langmodule] [voice=voicelib] [gain=db] [pitch=adjust] [speed=fast—slow] [volume=%level] [text=message] [extension=fileext] phrasebook-rules words...*
Used to implement phrasebook rules based prompt generation. The **altspeak** *version exists to speak a phrase only if there is no tts system installed, and is meant to be used after a* **say** *command. The current voice library and language module options may be used, or new ones may be specified on the command line for the current command only.*

start[.offset—.group—.ext—.trunk—.tie—.span—.wait] *[var=&symbol]-[maxTime=timeout] [submit=vars-to-copy] [expire] offset—group script [parms]*

start a script on another trunk port as offset from the current port %id or by issuing a request against another trunk group. Hence "start 1 test" starts script test on the port next to the current one, for example. Normally, a large offset like "start 24 test" might be used to start a script on the next T span. Start can start a script immediately, or time delayed as a queued request with the timeout specified in maxTime. A "var=&varname" can be used to save the started session id to a variable.

statinfo *id=groupname [capacity=&var] [incoming=&var] [outgoing=&var] [used=&var] [avail=&var]*
This command is used to collect active call statistics from a known trunk group entry. This can be used to determine how many calls are in process for a given group, for example. The specific named stat item entries are stored into specified variable names that are passed as part of keyword symbols.

sync.exit *[time=seconds]*
Set the exit timer for this call session based on the start time of the call. If no timeout is specified, then the exit timer is cleared. When an exit timer expires, and there is no t̂ime handler, the call session exits.

sync.start—.current *time=seconds]*
This is used to set a call event timer, which will invoke the t̂ime handler in a script, at a specified number of seconds from the original start time of the call, or for a number of seconds after the current time. If no time is specified, then any previously set timer is cleared.

sync *time=duration [maxRing=rings]*
Sleep the current call until the total time since the start of call is equal to the specified duration. This is like a sleep call, but scheduled from start of the call rather than from the current time. A version of sleep.start may be added later to also do this.

tone *[count=repeat] [timeout=intertone] name*
Play a named tone, as defined in the bayonne.conf file. The tone can be repeated a specified number of times.

tone *[count=repeat] [timeout=intertone] frequency=freq [amplitude=amp] length=duration*
Play a dynamically constructed monofrequency tone on the fly.

tone *[count=repeat] [timeout=intertone] freq1=freq freq2=freq [ampl1=amp] [ampl2=amp] length=duration*
Play a dynamically constructed dual frequency tone on the fly.

11.8 Bayonne Preference and User Session Management

These commands deal with management of various persistant databases and the concept of user sessions in Bayonne. User sessions are based on the idea that a given call session may be logged in under a specified login id. Login id's may be used for any purpose, including PBX extensions, voice mailboxes, pins for debit systems, etc.

change *id=usertag value=newvalue*
Change a user property in the preferences database for the currently logged in user to the specified value. The actual change is stored with **commit**.

commit
Commit may be used to commit changes made to user preferences through the change or password command so that they are permenantly stored. This only is effective when logged in under a user id session.

huntinfo *id=pilot tag=&var ...*
While hunt groups are primarily used in scripting PBX systems, they could be used generically in bayonne for other purposes. A hunt group is a persistant data record under a known pilot number who's script defined tags and values may be extracted with a simple script interface.

login *id=userid password=password*
Attempt to set the current call session under a specified user id. This will be successfull if the correct password is used.

logout
Logs out of the current active user session.

password *[id=userid] password*
This is used to change the password of the preference and login for the currently logged in user id. If the current user is priviledged, it may change the password of other users as well.

reset *id=usertag*
Resets a user property in the preferences database for the currently logged in user to it's default. The actual change is stored with **commit**.

userinfo *id=userid infokey=&var*
This command is used to extract information about foreign user id's that are stored in the preference system. The currently logged in user's info is accessed

through %user.xxx.

11.9 Driver specific commands

The exact availability and behavior of a number of script commands do depend on specific features or capabilities that must be provided for by specific telephony drivers. These capabilities and features may not be universally available since some drivers will be missing features or capabilities that might be found in others. These driver specific Bayonne script commands are described here:

accept *label*
Some Bayonne cti drivers support ISDN (pri) hardware where call accepting may be manually controlled. When call accpetance is manually controlled, intercept messages may be played back to users, and these are unbilled. The billing clock is only enabled from the perspective of the telco if the call has been accepted, and this can be done with the accept command. The accept command then branches to a new script because the hangup handler of this type of script must use **reject** *to reject the call if he has hung up before it has been accepted.*

answer.intercom—.trunk—.parent—,pickup *label*
Drivers with PBX support have an enhanced version of the answer command that enables answer to be used as a reply to inter-call session intercom dialing and pickup requests. This is used to notify the intercom dialer or pickup requestor that an answer response has occured on a given extension or line. On successful answer completion, a branch is taken.

dial.intercom *ringback=tone count=rings [transfer=referer] [name=display] station ...*
Drivers capable of PBX support introduce a new and special form of the dial command. This special dial is used for intercom dialing, and supports the idea of being in a special intercom dial state. The extension engaged in intercom dialing receives an artificial ringback signal, and each station in the possible list of stations is dialed until it is picked up, or the specified count of rings has been waited.

join[.hangup] *id=port [waitTime=retrytimer] [maxTime=totallimit]*
This is the join command represented in most telephony drivers which have either TDM support or the ability to do soft joins. Join attempts to connect the channel to the port specified in a private full duplex conversation. If .hangup is used, then both ends hang up (exit) when the join completes. A join is normally a one time join attempt, but may be retried over a time interval if a waittimer is specified.

The value of %script.error is set based on the reasons that join was parted. The .parent option refers to the call session that started this one if it happened through a start command. Other options, such as .pickup and .intercom, are only available in PBX capable drivers.

join.parent—.transfer—.pickup—.recall *[waitTime=retrytimer] [maxTime-=totallimit]*
This version of join joins to trunk call state session identifiers. .parent refers to the %session.parent, while the others are are used in PBX drivers only.

pickup.incoming—.hold—.intercom—.trunk—.parent—.recall *label*
Pickup is used to send messages to a specified port identifer or incoming call source that are translated to p̂ickup handlers which may then be answered or joined to. This can be used to interupt a voice mail session when the station user originally called picks up the line, for example. This feature is only available in PBX drivers.

reject
This is used in conjunction with the **accept** *command in a script, and is usually used as a ĥangup handler, as it rejects the call, terminating it without billing. See* **accept** *for further details.*

ring.start *[group=trunkgroup] [source=othercallsession] extensionids...*
This is a PBX driver feature to initiate station ringing on behalf of another trunk or station. Source can be used to indicate the ring is being started on behalf of another station, and hence acts as an alternate means of doing a blind transfer. A trunk or port group may also be used or referenced to do ringing for a group.

ring.stop *[group=trunkgroup] extensions...*
This is a PBX driver feature to turn of station ringing for the specified stations.

ring.clear
This is a PBX driver feature to clear all pending ring requests against the current station.

wait[.hangup] *[id=session] maxTime=waittimer*
This is the generic version of wait used by all drivers with TDM or soft join support, A station that is waiting can wait for a join either from any station that attempts to join it, or from a specific station. It can also wait up to a specific time interval for the join to occur. The .hangup option is used to indicate hangup will occur when the join is over. Otherwise %script.error will hold the reason that the join ended.

wait.parent—.transfer—.pickup—.recall *maxTime=retrytimer*
This version of wait waits for a join from a specific trunk by the call state session identifiers. .parent refers to the %session.parent, while the others are are used in PBX drivers only.

11.10 XML support based plugins

There are a number of Bayonne script commands that are based on or involved with the presense of XML support, when it has been enabled for bayonne. A number of specialized plugins also exist and may be used only when XML support has been enabled. The effected script commands include:

assign *var=name [size=bytes] [value=value]*
While the assign command is built into Bayonne regardless of whethere XML support is enabled or not, it is most often used for supporting XML plugins which contain scripting languages that need to set or modify session symbols. It may be viewed as an alternative to **set**.

bayonnexml *url=http:xxxx [submit=vars] [maxTime=timeout]*
When the BayonneXML plugin is installed, the bayonnexml command can be used as a convenient shortcut script command that sets the xml parser to the BayonneXML dialect, and then performs a http "get" request to retrieve a BayonneXML document from a web server, using the query variables passed in submit. If such a document is successfully retrieved and parsed, then it is executed.

dir[.reverse] *prefix=subdir [var=&count] [extension=fileext] [match=prefix]*
The directory plugin creates a special **dir** *command which can be used to scan the contents of a specified subdirectory from /var/lib/bayonne. The contents are returned as multi-row data that can then be examined with the* **read** *command. The columns returned includes filename, sample size, and annotation for audio files. Matches can be done by name prefix and/or by specific file extensions. A variable can be specified to receive the total directory count.*

sql *[query=string] [maxTime=timelimit] query...*
Issue a SQL query through the selected sql plugin driver. The driver returns a multi-row data result that can then be examined with the "read" command. The data tuples are in table source "#sql" and the header can be read from "#header".

12 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.

13 Phrasebook Rules

13.1 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:

speak &balance %balance nomoney

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

13.2 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 the week of a date.*
&weekday	*speak the current day of the week.*
&time	*speak a time.*
&primary	*speak primary currency value (dollar(s) and cent(s))*
&local	*speak local currency*
&duration	*speak hours, minutes, and seconds, for duration values.*
&cy	*speak default currency (either primary, local, or both)*

13.2.1 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

13.2.2 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

13.2.3 Date/Time Prompts

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

13.2.4 Currency Prompts

dollar, dollars, cent, cents, and, or

14 Copyright