ChatScript Command Line Parameters

Revision 10/16/2022 cs12.3

Command Line Parameters

You can give parameters on the run command or in a config file or via a http request. The default config file is `cs_init.txt` at the top level of CS (if the file exists). A second file if present can add or override values - cs_initmore.txt And then third level named after the current language chosen can override those two cs_initenglish.txt being the default. The language one, in particular is useful to rebalance memory sizings (dictionary entries in particular) since foreign languages require more entries than English.

Or you can name where the file is on a command line parameter `config=xxx`. And `config2=xxx` for the initmore file. If you have secret information that you don't want stored in a config file or exposed on a command line then you can request the config data from a URL. Use the command line parameter configurl=http://xxx to specify the address of the data. Additional command line parameters, configheader=xxx, can be included to define HTTP request headers. If there are several headers then use separate configheader=xxx configheader=yyy etc parameters for each header name/value pair.

Config file data are command line parameters, 1 per line, like below:

```
noboot
port=20
```

Some parameters require a value and use the `=` format with no spaces. Other parameters may only require you name the parameter (they have no choices of values).

Actual command line parameters have priority over config file values, and those have priority over http requested values.

Language options

The command line parameter `language=` as shipped would be

```
language=english
```

but it can consist of a series of languages separated by commas with the first language being the default until you use a command to change to some other language on the fly. This enables multi-dictionary behavior.

```
language=english,spanish,german,japanese
```

When compiling your scripts, you may want different things in different languages. For example here is a multi-language level 0 script that compiles std cs concepts simultaneously in different languages.

```
language: UNIVERSAL
RAWDATA/ONTOLOGY/       # underlying vocabulary organization
language: ENGLISH
RAWDATA/ONTOLOGY/ENGLISH//
RAWDATA/WORLDDATA/
language: GERMAN
RAWDATA/ONTOLOGY/GERMAN//
language: SPANISH
RAWDATA/ONTOLOGY/SPANISH//
```

Memory options

Chatscript statically allocates its memory and so (barring unusual circumstance) will not allocate memory ever during its interactions with users. These parameters can control those allocations. Done typically in a memory poor environment like a cellphone.

option	description
`buffer=50`	how many buffers to allocate for general use (80 is default)
`buffer=15x80`	allocate 15 buffers of 80k bytes each (default buffer size is 80k)
`fullinputlimit=150000`	the input buffer(default buffer size is 80k)

Most chat doesn't require huge output and buffers around 20k each will be more than enough. 20 buffers is often enough too (depends on recursive issues in your scripts).

If the system runs out of buffers, it will perform emergency allocations to try to get more, but in limited memory environments (like phones) it might fail. You are not allowed to allocate less than a 20K buffer size.

option	description
`dict=n`	limit dictionary to this size entries
`text=n`	limit string space to this many bytes
`fact=n`	limit fact pool to this number of facts
`hash=n`	use this hash size for finding dictionary words (bigger = faster access)
`cache=50x1`	allocate a 50K buffer for handling 1 user file at a time. A server might want to cache multiple users at a time.

A default version of ChatScript will allocate much more than it needs, because it doesn't know what you might need.

If you want to use the least amount of memory (multiple servers on a machine or running on a mobile device), you should look at the USED line on startup and add small amounts to the entries (unless your application does unusual things with facts).

If you want to know how much, try doing `:show stats` and then `:source REGRESS/bigregress.txt`. This will run your bot against a wide range of input and the stats at the end will include the maximum values needed during a volley. To be paranoid, add beyond those valuse. Take your max dict value and double it. Same with max fact. Add 10000 to max text.

Just for reference, for our most advanced bot, the actual max values used were: max dict: 346 max fact: 689 max text: 38052.

And the maximum rules executed to find an answer to an input sentence was 8426 (not that you control or care). Typical rules executed for an input sentence was 500-2000 rules. For example, add 1000 to the dict and fact used amounts and 10 (kb) to the string space to have enough normal working room.

Output options

option	description
`output=nnn`	limits output line length for a bot to that amount (forcing crnl as needed). 0 is unlimited.
`outputsize=80000`	is the maximum output that can be shipped by a volley from the bot without getting truncated.

Actually the `outputsize` value is somewhat less, because routines generating partial data for later incorporation into the output also use the buffer and need some usually small amount of clearance. You can find out how close you have approached the max in a session by typing `:memstats`. If you need to ship a lot of data around, you can raise this into the megabyte range and expect CS will continue to function. 80K is the default. For normal operation, when you change `outputsize` you should also change `logsize` to be at least as much, so that the system can do complete logs. You are welcome to set log size lots smaller if you don't care about the log.

File options

option	description
`livedata=xxx`	name relative or absolute path to your own private LIVEDATA folder. Do not add trailing / on this pathRecommended is you use `RAWDATA/yourbotfolder/LIVEDATA` to keep all your data in one place. You can have your own live data, yet use ChatScripts default `LIVEDATA/SYSTEM` and `LIVEDATA/ENGLISH` by providing paths to the `system=` and `english=` parameters as well as the `livedata=` parameter
`topic=xxx`	name relative or absolute path to your own private TOPIC folder. Do not add trailing / on this path `/`
`buildfiles=xxx`	name relative or absolute path to your own folder where filesxxx.txt is found. Do not add trailing / on this path `/`
`users=xxx`	name relative or absolute path to where you want the USERS folder to be. Do not add trailing `/`
`logs=xxx`	name relative or absolute path to where you want the LOGS folder to be. Do not add trailing `/`
`userlog`	Store a user-bot log in USERS directory (default)
`userlog=1`	alternate form of request, you can use 0 for off and 1 for on (file). 2 means stdout 4 means stderr - you may combine
`userlogging=file`	primary form of request, you can use none for off and file, stdout, stderr. - you may combine userlogging=file,stdout
`nouserlog`	Don't store a user-bot log
`serverlog`	Store a server log in LOGS directory (default)
`serverlog=1`	alternate form of request, you can use 0 for off and 1 for on (file). 2 means stdout 4 means stderr - you may combine
`serverlogging=file`	primary form of request, you can use none for off and file, stdout, stderr. - you may combine serverlogging=file,stdout
`noserverlog`	Don't store a server log
`noretrybackup`	Don't save volley backup files for :retry when in standalone mode
`tmp=xxx`	name relative or absolute path to where you want the TMP folder to be. Do not add trailing `/`
`crashpath=xxx`	file to write about fatal Linux signals that will be outside of the cs folder `/`
`windowsbuglog=xxx`	names a WINDOWS directory to replicate the BUGS.txt log file outside of the CS directory area
`linuxsbuglog=xxx`	names a LINUX directory to replicate the BUGS.txt log file outside of the CS directory area
`Vcs_new_user=if input`	if input has given text within it, treat user as new and dont read the topic file
`deploylogging=delay=5`	disable server logging for 5 minutes after a deploy before using default logging value

Execution options

option	description
`source=xxxx`	Analogous to the `:source` command. The file is executed
`login=xxxx`	The same as you would name when asked for a login, this avoids having to ask for it. Can be `login=george` or `login=george:harry` or whatever
`build0=filename`	runs `:build` on the filename as level0 and exits with 0 on success or 4 on failure
`build1=filename`	runs `:build` on the filename as level1 and exits with 0 on success or 4 on failure.Eg. ChatScript `build0=files0.txt` will rebuild the usual level 0
`debug=:xxx`	xxx runs the given debug command and then exits. Useful for `:trim`, for example or more specific `:build` commands
`param=xxxxx`	data to be passed to your private code
`bootcmd=xxx`	(see Advanced Layers manual)
`trace`	turn on all tracing.
`redo`	see documentation for :redo in ChatScript Debugging Manual manual
`noboot`	Do not run any boot script on engine startup
`logsize=n`	When server log file exceeds n megabytes, rename it and start with a new file.
`defaultbot=name`	overrides defaultbot table for what bot to default to
`inputlimit=n`	truncate user input line to this size
`trustpos`	obey word~n and other pos restrictions in keywords
`autoreload`	in event of cs engine crash, output a dummy message and reload on next input (see $cs_crashmsg and $cs_crash)
`nofastload`	If you suspect fast loading is faulty, you can set this to see if things work without it
`syslogstr=xxx`	In linux will output this as part of Microsoft sql trace data to the syslog
`buildflags=xxx`	this data will be used to control :build (quiet and nomixedcase are xxx values)
`autorestartdelay=n`	in event of cs engine internal restart, delay n milliseconds before accepts users
`crnl_safe`	tells system it does not need to search for cr or nl to remove from inputs.
`blockapitrace`	disables any %trace_on in ^testpattern and ^testoutput. Used for production servers.
`traceboot`	turns on tracing while cs_boot is running at startup
`parselimit=n`	if input is larger than n characters, disable intense spellchecking, pos-tagging, and parsing for speed
`nl_save=1`	for ^testpattern, enables results of nl analysis to be saved onto the $cs_nlinfo variable so that outside can pass it back in on future calls

option	description
`parselimit=1`	inputs longer than this will get no pos-tagging,parsing, or spellchecking - speeds up
`random=n`	will force a specific value to be returned from %random

Trustpos is normally off by default because CS is only about 94% accurate in its built-in pos-tagging. So it prefers to wrongly match by allowing all pos values Of a word rather than miss a match. Ergo concept: $_{\text{all(feel}}$n) will match any use of "feel" rather than just noun meaning. But combining CS with Treetagger for english (if you license it) is better at pos-tagging than either alone, making it competitive with the best taggers in the world.

Here few command line parameters usage examples of usual edit/compile developement phase, running ChatScript from a Linux terminal console (standalone mode):

Rebuild *level0* (compiling system ChatScript files, listed usually in file `files0.txt`):

```
BINARIES/LinuxChatScript64 local build0=files0.txt
```

Rebuild *level1*, compiling bot *Mybot* (files listed in file `filesMybot.txt`), showing build report on screen (stdout):

```
BINARIES/LinuxChatScript64 local build1=filesMybot.txt
```

Rebuild and run: If building phase is without building errors, you can run the just built *Mybot* in local mode (interactive console) with user name *Amy*:

```
BINARIES/LinuxChatScript64 local login=Amy
```

Build bot *Mybot* and run ChatScript with user *Amy*:

```
BINARIES/LinuxChatScript64 local build1=filesMybot.txt &&  BINARIES/LinuxChatScript64 local
```

Bot variables (aka Server variables)

You can create predefined bot variables by simply naming permanent variables on the command line, using V to replace $ (since Linux shell scripts don't like $). Eg.

```
ChatScript Vmyvar=fatcat
```

```
ChatScript Vmyvar="tony is here"
```

```
ChatScript "Vmyvar=tony is here"
```

Quoted strings will be stored without the quotes. Bot variables are always reset to their original value at each volley, even if you overwrite them during a volley. This can be used to provide server-host specific values into a script. Nor will

they be saved in The user's topic file across volleys. This also applies to variables defined during any CS_BOOT

No such bot-specific - nosuchbotrestart=true

If the system does not recognize the bot name requested, it can automatically restart a server (on the presumption that something got damaged). If you don't expect no such bot to happen, you can enable this restart using `nosuchbotrestart=true`. Default is false.

Time options

option	description
`Timer=15000`	if a volley lasts more than 15 seconds, abort it and return a timeout message.
`Timer=18000x10`	same as above, but more roughly, higher number after the x reduces how frequently it samples time, reducing the cost of sampling
`Timelog=5000`	if a volley lasts more than 5000 milliseconds, record it in LOGS/time.txt and for LINUX in /log/cstime.txt .

`:TranslateConcept` Google API Key

option	description
`apikey=xxxxxx`	is how you provide a google translate api key to `:translateconcept`

Security

Typically security parameters only are used in a server configuration.

option	description
`sandbox`	If the engine is not allowed to alter the server machine other than through the standard ChatScript directories, you can start it with the parameter `sandbox` which disables Export and System calls.
`nodebug`	Users may not issue debug commands (regardless of authorizations). Scripts can still do so.

option	description
`authorize=""`	bunch of authorizations "". The contents of the string are just like the contents of the authorizations file for the server. Each entry separated from the other by a space. This list is checked first. If it fails to authorize AND there is a file, then the file will be checked also. Otherwise authorization is denied.
`encrypt=xxxxx decrypt=xxxx`	These are URLs that accept JSON data to encrypt and decrypt user data. User data is of two forms, topic data and LTM data. LTM data is intended to be more personalized for a user, so if `encrypt` is set, LTM will be encrypted. User topic data is often just execution state of the user and potentially big, so by default it is not encrypted. You can request encryption using `userencrypt` as a command line parameter to encrypt the topic file and `ltmdecrypt` to encrypt the ltm file.

The JSON data sent to the URL given by the parameters looks like this:

```
{"datavalues": {"x": "..."}}
```

where ... is the text to encrypt or decrypt. Data from CS will be filled into the ... and are JSON compatible.

Server Parameters

Either Mac/LINUX or Windows versions accept the following command line args:

option	description
`port=xxx`	This tells the system to be a server and to use the given numeric port. You must do this to tell Windows to run as a server. The standard port is 1024 but you can use any port.
`local`	The opposite of the port command, this says run the program as a stand-alone system, not as a server.
`interface=127.0.0.1`	By default the value is `0.0.0.0` and the system directly uses a port that may be open to the internet. You can set the interface to a different value and it will set the local port of the TCP connection to what you designate. 127 is the classic TCP port.

User Data

Scripts can direct the system to store individualized data for a user in the user's topic file in USERS. It can store user variables (`$xxx`) or facts. Since variables hold only a single piece of information a script already controls how many of those there are. But facts can be arbitrarily created by a script and there is no natural limit. As these all take up room in the user's file, affecting how long it takes to process a volley (due to the time it takes to load and write back a topic file), you may want to limit how many facts each user can have written. This is unrelated to universal facts the system has at its permanent disposal as part of the base system.

`userfacts=n` limits a user file to saving only the n most recently created facts of a user (this does not include facts stored in fact sets). Overridden if the user has `$cs_userfactlimit` set to some value

User Caching

Each user is tracked via their topic file in USERS. The system must load it and write it back for each volley and in some cases will become I/O bound as a result (particularly if the filesystem is not local).

You can direct the system to keep a cache in memory of recent users, to reduce the I/O volume. It will still write out data periodically, but not every volley. Of course if you do this and the server crashes, writebacks may not have happened and some system rememberance of user interaction will be lost.

Of course if the system crashes, user's may not think it unusually that the chatbot forgot some of what happened. By default, the system automatically writes to disk every volley, If you use a different value, a user file will never be more out of date than that.

```
cache=20
cache=20x1
```

This specifies how many users can be cached in memory and how big the cache block in kb should be for a user. The default block size is `50` (50,000 bytes). User files typically are under 20,000 bytes.

If a file is too big for the block, it will just have to write directly to and from the filesystem. The default cache count is 1, telling how many users to cache at once, but you can explicitly set how many users get cached with the number after the "x". If the second number is 0, then no caching is done and users have no data saved. They remember nothing from volley to volley.

Do not use caching with fork. The forks will be hiding user data from each other.

```
save=n
```

This specifies how many volleys should elapse before a cached user is saved to disk. Default is 1. A value of 0 not only causes a user's data to be written out every volley, but also causes the user record to be dropped from the cache, so it is read back in every time it is needed (handy when running multi-core copies of chatscript off the same port).

Note, if you change the default to a number higher than 1, you should always use `:quit` to end a server. Merely killing the process may result in loss of the most recent user activity.

Logging or Not

In stand-alone mode the system logs what a user says with a bot in the USERS folder. It can also do this in server mode. It can also log what the server itself does. But logging slows down the system. Particularly if you have an intervening server running and it is logging things, you may have no use whatsoever for ChatScript's logging.

`Userlog`

Store a user-bot log in USERS directory. Stand-alone default if unspecified. Alternatively you can do userlog=1 to enable.

`Nouserlog`

Don't store a user-bot log. Server default if unspecified. Alternatively you can do userlog=0 to disable. Alternatively you can do userlogging=none.

`Serverlog`

Write a server log and a bugs log. Alternatively you can do serverlog=1 to enable to file, 2 to use stdout, 4 to use stderr. You can combine like serverlog=5 Prefer the alternate form of request, serverlogging=none for off and file, stdout, stderr. - you may combine serverlogging=file,stdout Last form scanned (read in order from cs_init.txt and then cs_initmore.txt) wins.

The server log will be put into the LOGS directory under serverlogxxx.txt where xxx is the port.

The bugs log is in the same directory under bugs.txt (all ports).

The server log records all transactions by all users in order of arrival. Whereas the user log records transactions by user/bot.

The server log can be written regardless of whether CS is running in server mode or not.

`serverlogauthcode=xxxxx`

In addition to permanently turning on server and/or user logging, you can provide a cheat code in your input that, if it matches the serverlogauthcode,

will enable server and user logging for that input. This code is hidden from CS processing so it will not impact NL processing.

In calls to ^testpattern, this will log to return in a `trace` field. By default, this code will just trace patterns. But if you concat `2` onto the code it will do a full cs trace.

In calls to ^testouput, this will log to return in a trace field but will always be a full trace.

`hidefromlog="label label2 label3"`

If there is data you don't want reflected into either server or user log files, this is the parameter. Maybe you don't want an authorization code recorded, or whatever. This presumes the data is part of some JSON object. You name one or more labels and when those are found in data outbound to a log file, the label and its value will be omitted.

`Serverctrlz`

Have server terminate its output with 0x00 0xfe 0xff as a verification the client received the entire message, since without sending to server, client cannot be positive the connection wasn't broken somewhere and await more input forever.

`pseudoserver`

This asserts that cs in DLL/sharedobject form is being used as a server (though its caller is doing all the server work). This enables the required authorizations from CS before using debug commands.

`Noserverlog`

Superceeded by `serverloggging=`. Don't write a server log or a bugs log. Alternatively you can do serverlog=0 to disable.

`Nobuglog`

Don't write a bugs log. Same as Buglog=0 or Buglog=none

`Buglog`
`Buglog=n`
`Buglogging=none,file,stdout,stderr`

Write a LOGS/bugs.txt log if n = 1. Write nothing if n == 0. Alternate form of request, you can use buglogging=none for off and file, stdout, stderr. - you may combine buglogging=file,stderr

`DebugLevel=n`

Sets debug level of server logging. 0 will remove logging all the startup variables and their values.

`Fork=n`

If using LINUX EVSERVER, you can request extra copies of ChatScript (to run on each core for example). n specifies how many additional copies of ChatScript to launch.

`Serverretry`

Allows `:retry` to work from a server - don't use this except for testing a single-person on a server as it slows down the server.

`servertrace`

when present, forces all users to have tracing turned on. Traces go to the user logs.

`repeatLimit=n`

Servers are subject to malicious inputs, often generated as repeated words over and over. This detects repeated input and if the number of sequential repeats is non-zero and equal or greater to this parameter, such inputs will be truncated to just the initial repeats. All other input in this volley will be discarded.

`erasename=myname`

:reset, when called from running script, is unable to fully reset the system. Facts that have already been created are not destroyed and user variables that have been defined are not erased, only ones in the bot definition are changed back to their default settings. The erasename parameter is used to perform a full reset prior to loading the user topic file. The incoming input is scanned for the text given, and if found the system bypasses loading the topic file and instead just initializes a fresh bot. The actual erasename seen in input will be converted to all blanks, so it will not disturb normal behavior, either in OOB input or user input.

The default value for this is: `csuser_erase` which you can change to anything else you want.

`cs_new_user="text""`

The cs_new_user parameter is used to perform a full reset prior to loading the user topic file. The incoming input is scanned for the text given, and if found the system bypasses loading the topic file and instead just initializes a fresh bot. This differs from erasename in that the text is not erased.

No such bot-specific - nosuchbotrestart=true

If the system does not recognize the bot name requested, it can automatically restart a server (on the presumption that something got damaged). If you don't expect no such bot to happen, you can enable this restart using `nosuchbotrestart=true`. Default is false.

Testing a server

There are various configurations for having an instance be a client to test a server.

`client=xxxx:yyyy`

This says be a client to test a remote server at IP xxxx and port yyyy. You will be able to "login" to this client and then send and receive messages with a server.

`client=localhost:yyyy`

This says be a client to test a local server on port yyyy. Similar to above.

`Load=1`

This creates a localhost client that constantly sends messages to a server. Works its way through `REGRESS/bigregress.txt` as its input (over 100K messages). Can assign different numbers to create different loading clients (e.g., load=10 creates 10 clients).

`Dual`

Yet another client. But this one feeds the output of the server back as input for the next round. There are also command line parameters for controlling memory usage which are not specific to being a server.