ChatScript Engine and Private Code Manual
© Bruce Wilcox, gowilcox@gmail.com brilligunderstanding.com
Revision 5/18/2016 cs6.5a

This does not cover ChatScript the scripting language. It covers how the internals of the engine work and how to extend it with private code.

Data

First you need to understandthe basic data available to the user and CS and how it is allocated.

Text Strings

The first fundamental datatype is the text string. This is a standard null-terminated C string represented in UTF8. Text strings represent words, phrases, things to say. They represent numbers (converted on the fly when needed to float and int64 values for computation). They represent the names of functions and concepts and topics and user variables. Strings are allocated out of either the stack or the heap.

Dictionary Entries

The second fundamental datatype is the dictionary entry. A dictionary entry refers to a string and may have other data attached. Function names in the dictionary tell you how many arguments they require, where to go to execute their code. User variable names in the dictionary store a pointer to their value (in the stack or heap). Ordinary English words have bits describing their part-of-speech information and how to use them in parsing.

In fact, ordinary words have multiple meanings which can vary in their part of speech and their ontology, so the list of possible meanings and descriptions is part of the dictionary entry. When working directly with a dictionary entry, the code uses *WORDP* as its datatype (pointer to a word). Code working indirectly with a dictionary entry uses *MEANING* as its datatype. A *MEANING* is an index into the dictionary along with description bits that can say which meaning of the word is referred to or which part of speech form (like noun vs verb) is being referred to. In a concept definition, for example, one can refer to *break*, which means any meaning or POS-type of the word *break*. One refers to the 33rd meaning of *break* as *break~33*, which implies a specific part of speech and place in the ontology of the dictionary. One refers to noun meanings of break as *break~n*.

Facts

The third fundamental datatype is the fact, a triple of data. The fields are called *subject*, *verb,* and *object* but that is just a naming convention. They can hold any kind of data. Each field of a fact is either a MEANING (which in turn references a text string) or a direct index reference to another fact. As a direct reference, it is not text. It is literally an offset into fact space. But facts have description bits on them, and one such bit can say that a field is not a normal MEANING but is instead a binary number fact reference. While fact references in a fact field are binary numbers, when stored in variables, a fact reference is a text number string like *250066*. When stored in external files (like user long-term memory or ^export) facts are stored as full text describing the fact, e.g., *(bob like (apples and oranges) 0x10000*) which in this case is a fact with a fact as object. Externally facts are stored this way because user facts get relocated every volley and cannot safely be referred to directly by a fact reference number.

JSON Facts

CS directly supports JSON and you can manipulate JSON arrays and objects as you might expect. Internally, however, JSON is represented merely by facts with special bits that indicate how to use that JSON fact. JSON objects and arrays are represented as synthesized text strings like *jo-5* or *ja-1025*. Each key-value pair of the object is a fact like *(jo-5 location Seattle)*. Each array element pair is a fact like *(ja-5 1 dog)*. Array facts start with 0 as the verb and are always contigiously sequential (no missing numbers). Should you delete an element from the middle, all later elements automatically renumber to return to a contiguous sequence.

User Variables

User variables are names that can be found in the dictionary and have entries pointing to text strings. $ and $$ variables point to strings in the heap. $_ variables point to strings in the stack.

Match Variables

Match variables (_0) hold text strings directly. They use prereserved spaces and thus have limited sizes of strings they can hold. They are expecting to hold parts of a sentence that have been matched and sentences are limited to 254 words (and words are typically small). But a match variable is more complex than a user variable because usually a match variable holds extra data from an input sentence. It holds the original text seen in the sentence, the canonical form of that value, and the position reference for where in the sentence the data came from. Match variables pass all that information along when assigned to other match variables, but lose all but one of them when assigned to a user variable or stored as the field of a fact.

Finer details about words

Words are the fundamental unit of information in CS. The original words came from WordNet, and then were either reduced or expanded. Word are reduced when some or all meanings of them are removed because they are too difficult to manage. “I”, for example, has a Wordnet meaning of the chemical iodine, and because that is so rare in usage and causes major headaches for ChatScript, that definition has been expunged along with some 500 other meanings of words. Additional words have been added, including things that Wordnet doesn't cover like pronouns, prepositions, determiners, and conjunctions. And more recent words like “animatronic” and “beatbox”. Every word in a pattern has a value in the dictionary. Even things that are not words, including phrases, can reside in the dictionary and have properties, even if the property is merely that this is a keyword of some pattern or concept somewhere.

Words have zillions of bits representing language properties of the word (well, maybe not zillions, but 3x64 bytes worth of bits). Many are permanent core properties like it can be a noun, a singular noun, it refers to a unit of time (like “month”), it refers to an animate being, it's a word learning typically in first grade. Other properties result from compiling your script (this word is found in a pattern somewhere in your script). All of these properties could have been represented as facts, but it would have been inefficient in either cpu time or memory to have done so.

Some dictionary items are “permanent”, meaning they are loaded when the system starts up, either from the dictionary or from data in layer 0 and layer 1. Other dictionary items are “transient”. They

come into existence as a result of user input and will disappear when that volley is complete. They may live on in text as data stored in the user's topic file and will reappear again during the next volley when the user data is reloaded. Words like *dogs* are not in the permanent dictionary but will get created as transient entries if they show up in the user's input.

The dictionary consists of WORD entries, stored in hash buckets when the system starts up. The hash code is the same for lower and upper case words, but upper case adds 1 to the bucket it stores in. This makes it easy to perform lookups where we are uncertain of the proper casing (which is common because casing in user input is unreliable). The system can store multiple ways of upper-casing a word.

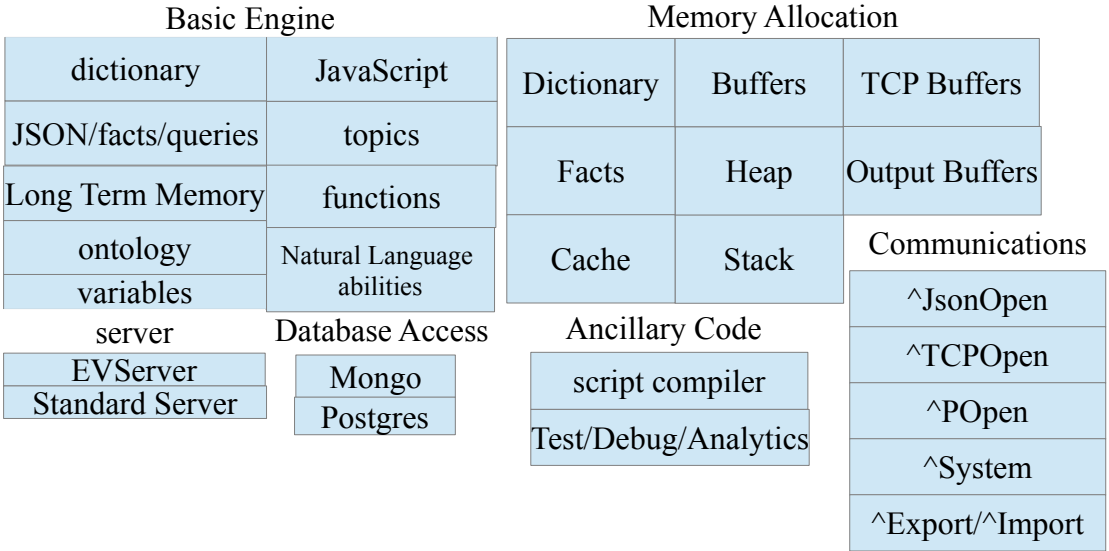
Facts are simply triples of words that represent relationships between words. The ontology structure of CS is represented as facts (which allows them to be queried). Words are hierarchically linked using facts (using the "is" verb). Word are conceptually linked (defined in a ~concept) using facts with the verb "member". Word entries have lists of facts that use them as either subject or verb or object so that when you do a query like query(direct_ss dog love ?) CS will retrieve the list of facts that have dog as a subject and consider those. And all those values of fields of a fact are words in the dictionary so that they will be able to be queried. Queries like "^query(direct_v ? walk ?) function by having a byte code scripting language stored on the query name "direct_v". This byte code is defined in LIVEDATA (so you can define new queries) and is executed to perform the query. Effectively facts create graphs and queries are a language for walking the edges of the graph.

ChatScript support user variables, for considerations of efficiency and ease of reference by scripters. Variables could have been represented as facts, but it would have increased processing speed, local memory, and user file sizes, not to mention made scripts harder to read.

Memory Management

Many programs use malloc and free extensively upon demand. These functions are not particularly fast. And they lead to memory fragmentation, whereupon one might fail a malloc even though overall the space exists. ChatScript follows video game design principles and manages its own memory. It allocates everything in advance and then (with rare exception) it never dynamically allocates memory again, so it should not fail by calling the OS for memory. And you have control over the allocations upon startup via command line parameters.

This does not mean CS has a perfect memory management system. Merely that it is extremely fast. It is based on mark/release, so it allocates space rapidly, and at the end of the volley, it releases all the space it used back into its own pool. In the diagram below under *Memory Allocation* you have a list of all the areas of memory that are preallocated.



You might run out of memory allocated to dictionary items while still having memory available for facts. This means you need to rebalance your allocations. But most people never run into these problems unless they are on mobile versions of CS.
Stack and Heap memory are allocated out of a single chunk. Stack is at the low end of memory and grows upwards while Heap is at the high end and grows downward. If they meet, you are out of space.

Stack memory is tied to calls to *ChangeDepth* which typically represent function or topic invocation. Once that invocation is complete, all stack space allocated is cut back to its starting position. Stack space is typically allocated by *AllocateStack* and explicitly released via *ReleaseStack.* If you need an allocation but won't know how much is needed until after you have filled the space, you can use *InfiniteStack* and then when finished, use *CompleteBindStack* if you need to keep the allocation or just use *ReleaseStack* to discard it. While InfiniteStack is in progress, you cannot safely make any more *AllocateStack* calls.

Heap memory allocated by *AllocateHeap* lasts for the duration of the volley and is not explicitly deallocated. So conceivably some heap memory is free but hasn't been freed. But CS supports planning, which means backtracking, which means memory is really not free along the way because the system might revert things back to some earlier state. This problem of free memory mostly shows up in document mode, where reading long paragraphs of text are all considered a single volley and therefore one might run out of memory. CS provides ^*memorymark* and ^*memoryfree* so you can explicitly control this while reading a document.

Buffers are allocated and deallocated via *AllocateBuffer* and *FreeBuffer.* Typically they are used within a small block of short-lasting code, when you don't want to waste heap space and cannot make use of stack space. While there are a small amount of them preallocated, in an emergency if the system runs out of them it can malloc a few more.

TCP buffers are dynamically allocated (violating the principle of not using malloc/free) via accept threads of a server.

Output buffers refer to either the main user output buffer or the log buffer. The output buffer needs to be particularly big to hold potentially large amounts of OOB (out-of-band) data being shipped externally. Also most temporary computations from functions and rules are dumped into the output buffer temporarily, so the buffer holds both output in progress as well as temporary computation. So if your output were to actually be close to the real size of the buffer, you would probably need to make the buffer bigger to allow room for transient computation. The log buffer typically is the same size so one can record exactly what the server said. Otherwise it can be much smaller if you don't care.

Cache space is where the system reads and writes the user's long term memory. There is at least 1, to hold the current user, but you can optimize read/write calls by caching multiple users at a time, subject to the risk that the server crashes and recent transactions are lost. A cache entry needs to be large enough to hold all the data on a user you may want saved.

Function Run-time Model

The fundamental units of computation in ChatScript are functions (system functions and user outputmacros) and rules of topics. Rules and outputmacros can be considered somewhat interchangeable as both can have code and be invoked (rules by calling ^reuse).

Function names are stored in the dictionary and either point to script to execute or engine code to call, as well as the number of arguments and names of arguments.

System functions are predefined C code to perform some activity most of which take arguments that are evaluated in advance (but use *STREAMARG* and wait until they get them to decide whether to evaluate or not). System functions can either designate exactly how many arguments they expect, or use *VARIABLE_ARGUMENT_COUNT* to allow unfixed amounts.

Outputmacros are scripter-written stuff that CS dynamically processes at execution time to treat as a mixture of script statements and user output words. They can have arguments passed to them as either call by value or call by reference.

There are actually two styles of passing arguments. Arguments are stored in a global argument array, referenced by ARGUMENT(n) when viewed from system routines. The call sets the current global index and then stores arguments relative to that. Outputmacros use that same mechanism for call-by-reference arguments. Call by reference arguments start with ^ like ^*myarg* and the script compiler compiles the names into number references starting with ^0 and increasing ^1 ... ^*myarg* style allows a routine to assign indirectly to the callers variable.

But outputmacros also support call by value arguments which start with *$_* like *$_myarg*. $_xxx style do not allow this because they might be receiving a caller's local $_ variables, and no one outside the routine is allowed to change them. These are normal albeit transient variables so the corresponding value from the argument stack is also stored as the value of the local variable. That happens after the function call code first saves away the old values of all locals of a routine (or topic) and then initializes all locals to NULL. Once the call is finished, the saved values are restored.

When passing data as call by value, the value stored always has a `` prefix in front of it. In fact, all assignments onto local variables have that prefix prepended (hidden). This allows the system to detect that the value comes from a local variable or an active string and has already been evaluated. Normally, if the output processor sees $xxx in the output stream, it would attempt to evaluate it. But if it looks and sees there is a hidden `` before it, it knows that $xxx is the final value and is not to be evaluated further.

` is a strongly reserved character of the engine and is prevented from occurring in normal data from a user. It is used to mark data coming preevaluated. It is used to mark ends of rules in scripts. It is used to quote values of variables and fact fields when writing out to the user's topic file.

Script Execution

The engine is heavily dependent upon the prefix character to tell the system how to process script. The script compiler normally forces separate of things into separate tokens to allow fast uniform handling. E.g., "^call(bob hello)" becomes "^call (bob hello)". This predictability allows the system to avoid all the logic involved in knowing where some tokens end and others begin. The other trick the script compiler uses is to put in characters indicating how far something extends. This jump value is used for things like *if* statements to skip over failing segments of the if. Actual script execution, be it output processing or pattern processing jumps via switch statements on the initial character of a token.

Evaluation Contexts

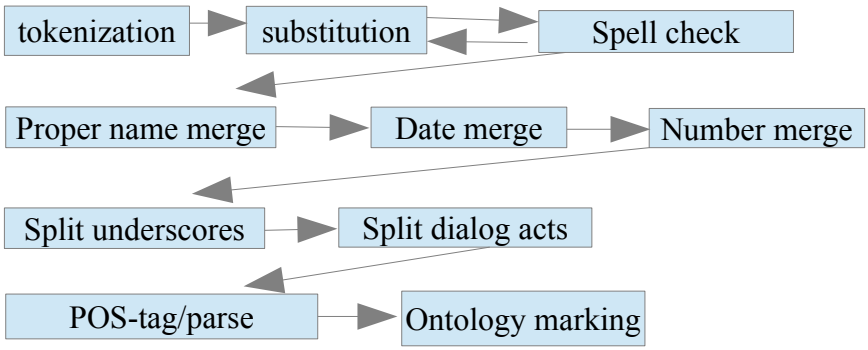
The evaluation contexts are

•outputSystem.cpp output function
•performAssignment left side
•if testing
•activeString ReformatString
•all STREAM argument engine functions
•DoFunction function call argument evaluation

Many things evaluate the same way. Match variables (_0) evaluate to their content. User variables evaluate to their content, unless they are dotted or subscripted, in which case they evaluate to their content and then perform a JSON object lookup.

Natural Language Pipeline

Aside from a range of script-callable engine functions that are oriented toward natural language processing and processing facts as part of reasoning in script, the system preprocesses user input to make it easy to find meaning within it. These are both classic and unusual NL processing tasks. Other than tokenization, which is required, and ontology marking, which is why you use ChatScript, the other stages are all completely script omittable under control of $cs_token.



Tokenization

The input is a stream of characters. If there is OOB data, this is detected and split off as a single word sentence in its own right. The rest of the input is then separated into the next sentence and the leftover unprocessed characters. A sentence has a 254 word limit, so anything attempting to be one beyond that will be split at that boundary. Tokenization tries to separate into normal words and special tokens, e.g. *I like (this).* tokenizes into *I like (this)* . It has a bunch of decisions to make around things like periods and commas. Is the period a sentence end or an abbreviation or web url or what? Is a comma a part of a number or a piece of punctuation of the sentence? If something is a *12/2/1933* date, it can be separated into *12 / 2 / 1933* which leaves the decision to date merge or not under control of the scripter.

While doing tokenization, some transformations revise toward canonical forms what is seen. *6'12"* is autoconvered into *6 feet 12 inches.*

The process of tokenization is not visible to the script. It cannot easily know what transformations were made. This becomes the real sentence the user input, and is visible from the ^original function.

Substitution

The LIVEDATA folder contains a series of files representing sequences of words one might expect and how you might want to revise them. Revisions are either to correct the user input or make it easier to understand the meaning by converting to a standard form. The files include:

british – a british spelling and its american equivalent, since the system is american based
contractions – expands contractions like *don't* into *do not.*
interjections – revises dialog acts like "so long." into ~emogoodbye or *"sure."* into ~yes
noise – removes useless words like *very* or *"I mean that"*
spellfix – common spelling mistakes that might be hard for spellcheck to fix correctly
substitutes – for whatever reason - e.g. *Britain* into *Great Britain*
texting – revise shorthands into normal words or interjections like *:)* into ~emohappy

Spell check

Standard spell check algorithms based on edit distance and cost are used, but only among words which are of the same length +- one.

The system also checks for bursting tokens into multiple words or merging multiple tokens into a word, and makes decisions about hypenated words. For english, since the dictionary only contains lemma (canonical) forms of words, it checks standard conjugations to see if it recognizes a word. For foreign languages, it lacks this code, so the dictionary needs to include all conjugated forms of a word.

Merging

One can have proper names merged, date tokens merged, and number tokens merged. Number-merging, in particular, converts things like *"four score and seven years ago"* into a single token *four_score_and_seven_years_ago* which is marked as a number and whose canonical value is *87.*

Splitting

Dialog acts can be split into separate sentences so that *Yes I love you* and *Yes, I love you* and *Yes. I love you* all become the same input of two sentences – the dialog act ~yes and *I love you*.

Pos-parsing

For English, which is native to CS, the system runs pos-parsing in two passes. The first pass is execution of rule from LIVEDATA/ENGLISH/POS which help it prune out possible meanings of words. The goal of these rules is to reduce ambiguity without ever throwing out actual possible pos values while reducing incorrect meanings as much as possible. The second pass tries to determine the parse of the sentence, forcing various pos choices as it goes and altering them if it finds it has made a mistake. It uses a "garden path" algorithm. It presumes the words form a sentence, and tries to directly find pos values that make it so in a simple way, changing things if it discovers anomolies.

For foreign languages, the system has code that allows you to plug in as a script call things that could connect to web-api pos-taggers. It also can directly integrate with the TreeTagger pos-tagger if you obtain a commerical license for one or more languages from them. Parsing is not done by TreeTagger so while you know part-of-speech data, you don't know roles like mainsubject, mainverb, etc.

Ontology Marking

The system takes the input and splits it into the original input and a canonical one. Both are "marked". Marking means taking the words of the sentence in order (where they may have pos-specific values) and noting on each word where they occur in the sentence (they may occur more than once). From specific words the system follows the *member* links to concepts they are members of, and marks those concepts as occuring at that location in the sentence. It also follows *is* links of the dictionary to determine other words and concepts to mark. And concepts may be members of other concepts, and so on up the hierarchy. There exist system functions that allow you, from script, to also mark and unmark words. This allows you to correct or augment meanings.

In addition to marking words, the system generates sequences of 5 contiguous words (phrases), and if it finds them in the dictionary, they too are marked.

Script Compiler

In large measure what the compiler does is verify the legality of your script and smooth out the tokens so there is a clean single space between each token. In addition, it inserts "jump" data that allows it to quickly move from one rule to another, and from an "if" test to the start of each branch so if the test fails, it doesn't have to read all the code involved in the failing branch. It also sometimes inserts a character at the start of a patttern element to identify what kind of element it is. E.g., = before a comparison token or * before a word that has wildcard spelling.

Private Code

You can add code to the engine without modifying its source files directly. To do this, you create a directory called *privatecode* at the top level of ChatScript. You must enable the PRIVATE_CODE define.

Inside it you place files:

- privatesrc.cpp - code you want to add to functionexecute.cpp (your own cs engine functions)
  - classic definitions compatible with invocation from script look like this:
    - static FunctionResult Yourfunction(char* buffer)
      - where ARGUMENT(1) is a first argument passed in.
      - answers are returned as text in buffer, and success/failure codes as FunctionResult.
- privatetable.cpp – listing of the functions made visible to CS
  - table entries to connect your functions to script:
  - { (char*) "^YourFunction", YourFunction, 1,0, (char*) "help text of your function"},
    - 1 is the number of evaluated arguments to be passed in
    - VARIABLE_ARGUMENT_COUNT means args evaled but you have to
      - detect the end – ARGUMENT(n) will be ?
    - STREAM_ARG – raw text sent. You have to break it apart and do whatever.
- privatesrc.h – header file. It must at least declare:
  - void PrivateInit(char* params); – called on startup of CS, passed param: private=
  - void PrivateRestart();- called when CS is restarting
  - void PrivateShutdown(); - called when CS is exiting.
- privatetestingtable.cpp – listing of :debug functions made visible to CS
  - Debug table entries like this:

{(char*) “:endinfo”, EndInfo,(char*)”Display all end information”},