ChatScript Javascript

Revision 4/30/2016 cs6.4

You can write functions in Javascript in your script files and invoke them from ChatScript code. It's all done with what you define in an `outputmacro:` construct.

Loading Javascript code

Normal outputmacros have a header (name + arguments in parens) and then ChatScript code. A Javascript definition replaces the ChatScript code part with a JavaScript header that tells CS what to use the data for, and then whatever JavaScript you want to write (optional).

That script will be uninterpreted (script compiler won't analyze it) up until the next top-level CS declaration starting some new line, or the end of file. The only analysis the script compiler will do is remove // comments that run to end of line. The script compiler will also replace leading whitespace on a line with a single blank, and remove carriage returns and newlines from source lines.

A Javascript header looks like this:

```
outputmacro: ^myfunction(^arg1) Javascript permanent eval
lines of javascript
```

The first word of the header is the word Javascript, which tells the system this is a Javascript declaration. The system currently handles Ecmascript E5/E5.1 e (Javascript) + some extensions from E6 (which was released in summer 2015). See http://duktape.org/guide.html for details.

The second word describes the context. CS maintains two different Javascript contexts. The first is `permanent`, lasting for the duration of the CS engine (server or local mode). The second is `transient`, which lasts during a volley only. Neither context can see data in the other context.

The third word describes whether to interpret the code upon execution of the outputmacro.

The javascript code is only executed when the outputmacro is invoked. And it will only be executed once no matter how many times you invoke the outputmacro. This is not a call to a javascript function (yet), this is how you declare code.

If you want to process your code into a context when building layer 0 or layer 1 (normal `:build` commands), you should arrange to invoke your output macro when the engine starts up by calling it from a `^csboot` function.

Loading javascript files

Instead of declaring lines of javascript locally, you can name files to load from. Just put

```
file filename
file filename
```

to load those files. e.g. this could have all the code from the earlier example

```
outputmacro: ^testx(^arg1) JavaScript permanent call void test int eval
file "test.js"
```

Invoking a function in JavaScript

In addition to establishing Javascript code as available, an outputmacro can define a linkage between CS and some function in Javascript. If we assume you have previously declared an outputmacro creating a bunch of javascript functions, one of which is called "test", then you could define a linkage to it via this:

```
outputmacro: ^func(^a1 ^a2 ^a3) Javascript permanent call void test string int float
```

The header is the same (Javascript permanent) and tells CS which context to find the function from. The call keyword defines the linkage by naming the return type to be used from Javascript routine, the name of the routine, and the types of the arguments to pass in to the javascript function. Every time this outputmacro is called, it will convert its arguments to be correct for Javascript, invoke the test function, and return the value appropriately.

Remember, before you can call your javascript code, you have to load it into the Javascript system. Which means calling once your outputmacro that defined your Javascript functions. You can avoid this with Declare and Use (below).

Declare and Use

You can simultaneously declare Javascript and declare a linkage. Just put eval on the javascript code after the linkage declaration. E.g.

```
outputmacro: ^func(^a1) Javascript permanent call void test string eval
bunch of javascript code
```

The first time you invoke the CS function, it will load the javascript into the designated context. And then it will perform the call. Future invocations of the CS function will not reload the javascript, merely perform the call.

Example:

```
outputmacro: ^testx(^arg1) JavaScript permanent call void test int eval

// fib.js
function fib(n) {
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
    return fib(n-1) + fib(n-2);
}

function test(j) {
     var res = [];
     for (i = 0; i < j; i++) {
     res.push(fib(i));
     }
 print(res.join(' '));
}

topic: ~keywordless()
u: (test) ^testx(5)
```

In the above example, if the `~keywordless` topic sees an input of `test` it will call `^testx` with a value of five. This will cause the javascript to be loaded and the javascript function `test` called with an integer value 5. This will, in turn, print out on the console `0 1 1 2 3` (but since that is not ChatScript output the system will move on to some other rule to try to generate output).

Notes & Restrictions

Currently the javascript is presumed correct (not verified by CS during `:build`). When the code is evaled, if there is an error, the rule making the outputmacro call will fail, but you have no indication of where in the javascript your code was in error.

Likewise there are nominal limits of 1500 bytes for a string argument or return value. These are not enforced at present, so you could damage the system if you exceed that. In the future restrictions and debuggability will be improved.

At present you can only pass in or return float, int, string. Any exception taken while running your script will end the ChatScript instance (this will get changed). Some obvious things... if you use print() you only see it on a local console.

On some OS compiles like IOS and Android, JavaScript support is by default not compiled into the engine. You would need to remove the appropriate #define DISCARDJAVASCRIPT 1 from common.h if you see as output something like `*JavaScript permanent...` instead of what you are expecting.