ChatScript Debugger Manual

© Bruce Wilcox, mailto:gowilcox@gmail.com www.brilligunderstanding.com Revision 4/8/2017 cs7.31

Debugging in and out

While it is possible to connect ChatScript to a GUI IDE for debugging (if you create one because CS does not come with one natively), CS also has a text command debugger analogous to GDB in LINUX.

You enter the debugger via

```
:debug
```

which begins a local command loop. Resume normal execution via

```
go
```

or

```
g
```

The debug command can also have one of the debug commands after it, e.g.

```
:debug break ^fn1 ~topic1 ^fn2
```

Variable and JSON displays

While in the debugger, you can display $variables just by typing in their name(s).

```
$myvar $_yourvar
```

If you know the name of a JSON unit, you can just type that and get a display of the array or object.

```
jo-44
```

Source locations

Typing the name of a function or topic will tell you the file and line number of it.

Breakpoints

You can set breakpoints that will trigger the debugger on a variety of conditions. And you can name multiple conditions at once. E.g.,

```
break ^myfunction ~mytopic ~mytopic.myrule
```

Function breakpoints

A function breakpoint will, by default, break on entry and break on exit. On entry it shows the function and its arguments. On exit it shows the value being returned and any error status.

You can restrict a function to only breaking on entry or exit by suffixing the name with @i or @o. Like

```
break ^myfunc@i
```

You can delete a breakpoint by putting `!` in front of the base name. E.g.

```
break ^myfunc !^otherfunc
```

Every time you call break, it prints out a summary of the breakpoints still set. So if you call break with no names, it just lists what you have.

Topic breakpoints

A topic breakpoint will break on entry and exit of topic by whatever manner (normal, `^gambit`, `^rejoinder`, `^reuse`, etc).

Rule breakpoints

A rule can broken upon by doing

```
break ~mytopic.label
```

where label is either the label of a rule or its tag (e.g., `1.0` for the 2nd top level rule of the topic).

Abnormal breakpoints

You can name

```
abort
```

to request a breakpoint if the system is about to exit due to abnormal conditions.

Variable Assigns

A different kind of breakpoint is a variable assign, that is, when a variable has its value changed. You can list multiple variables in a single request.

```
= $myvar  $_hisvar
```

The breakpoint will tell you the new value it is taking on. Be aware that if you break on a local variable like `$_myvar` then you break on the next change to it in whatever topic or function it is within. You may, if you want, create a restricted breakpoint for $_vars like this:

```
^myfunc.$_myvar
```

or

```
~mytopic.$_myvar
```

Clearing Breakpoints

```
clear
```

will remove all breakpoints including variable assigns.

Backtrace

```
where
```

will print out a backtrace of where you are in terms of topics, rules, and functions.

Controlled execution

You can step from action to action in your code by typing

```
s
```

or

```
step
```

Which is equivalent to `step across`. This will execute the next thing at the current level (maybe a function call, rule test, assignment or whatever). It will then display what code it executed, what value was returned, and what code would be executed next.

When you are at a rule, `s` moves to the next rule. Alternatively, you can ask the system to execute rules in a topic until it comes to one that would match by saying

`step match`

It will stop before entering the output of the rule or until it leaves the topic.

If you discover that the executed code called a function that did something wrong and you want to watch it, you can type

`redo in`

which will retry that same piece of code, but this time going in instead of stepping over.

You can step out from a function or topic by typing

`out`

which will execute code until it leaves the current function or topic or rule. topic is confusing when you have rules. If stepping out returns to code which has nothing left to do, that code will also complete and the system will pop out to the next level.

You can step in from an action in your code by typing

`in`

This will execute one or more bits of code until it calls into some function or exits the current scope entirely.

Executing script and other debug commands

You can type in a

`do`

command, which is analogous to `:do`, which allows you to set variables, execute script functions, etc.

Sourcing a file of debug commands

The command

`source filename`

will take debugger input from the named file (with a .txt suffix added to the name) where the file is in a top level folder called `debug`. Any command that resumes execution (like `go`, `in`, `out`, `s`) will terminate reading from the file.

In addition, you can create a file to automatically execute on a breakpoint. The name of the breakpoint should be the name of the file in the debug directory (with .txt added). When a function named the same as such a file has a breakpoint on

entry, it will execute the commands. Similarly whenever a topic has a breakpoint on entry, it will execute its commands.

Hooking up to a GUI IDE

The debugger can swap out its stdin and stdout interactions with the user with functions supplied on the `InitSystem` interface. The functions take a single argument `char*` and either get a line of input or write out a line of outout.

MAP file

The compiler builds a map file used by the debugger to know what files have what and what lines have what.

In addition, for every rule and function at the end of their data, the system prints out the `cyclomatic complexity` of the output code.