ChatScript Advanced Variable Manual

© Bruce Wilcox, gowilcox@gmail.com www.brilligunderstanding.com Revision 3/24/2019 cs9.2

Advanced Variables

Local Variables

User variables `$xxx` and `$$xxx` are all global in scope.

Anyone can access them to get or set them.

But if you want to write safe code, you also want local variables that no one can access and ruin on you. `$_xxx` are local variables.

When you use them inside an outputmacro or a topic, they only have meaning and access inside that code. No other code can see them. They can use the same variable name, and get their own local instance.

Local variables always start out initialized to null. If you pass them to an outputmacro, unlike normal variables which are passed by reference (hence can be written upon if you use Indirection Variables below), local variables are passed by value. Their content is passed, so no one can touch the variable itself.

Local variables used in a topic all remain accessible while in that topic. If you do `^gambit(~topic)` and then inside you do `^gambit(~)` or `^reuse(MYLABEL)`, you have not left the topic and so the variables remain intact. But if you call outside the topic, then a deeper call to the original topic sees new variables (just as a recursive call to an outputmacro would).

Hence `^gambit(~mytopic)` which calls `^reuse(~histopic.label)` which then calls `^gambit(~mytopic)` will get two different instances of `~mytopic` and the variables separately handled by each.

By definition, local variables are transient and do not get saved in a user's topic file.

Indirection Variables

You can, of course, merely refer to a variable in a script to set or get its value. But suppose you wanted to associate a variable with every topic you have.

Maybe when the user says *I'm bored* you want to leave a topic and not reopen it yourself for any near future (say 200 volleys).
In that case, you don't have a variable but you need to create one dynamically.

```
$$topicname = substitute(character %topic ~ "")
```

The above gives you a topic name without the ~. Pretend the current topic is ~washing. Then you can create a dynamic variable name simply by using

```
$$tmp = ^join($ $$topicname)
```

which would create the name $washing.

Now suppose you wanted to set that variable to some number representing the turn after which you might return.

```
$$tmp = 25
```

doesn't work. It wipes out the *washingvalueof*‘$tmp`and replaces it with`25‘.

You can set indirectly through `$$tmp` using function notation `^$$tmp` The above says take the value of `$$tmp`, treat it as the name of a variable, and assign into it. Which means it does the equivalent of

```
$washing = 25
```

If you want to an indirect value back, you can't do:

```
$$val = $$tmp
```

because that just passes the name of `$washing` over to `$$val`. Instead you do indirection again:

```
$$val = ^$$tmp # $$val becomes 25
```

Indirection works with values that are user variables and with values that are match variables or quoted match variables.

Many routines automatically evaluate a variable when using it as an argument, like `^gambit($$tmp)`. But if you want the value of the variable it represents, then you need to evaluate it another time.

```
$$tmp1 = ^$$tmp
^gambit($$tmp1)
```

Equivalently `^gambit(^$$tmp1)` is legal.

See also Indirect Pattern Elements in ChatScript Pattern Redux manual.

Bot variables

You can also define variables that are "owned" by the bot. Any variables you create as part of layer 0 or layer 1 are always resident. As are any variables you define on the command line starting ChatScript. You can see them and even change them during a volley, but they will always refresh back to their original values at the start of the next volley.

Match Variables

Match variables like `_5` are generally the result of using an underscore in a pattern match. Match variables hold 3 pieces of data

- original word(s)
- canonical word(s)
- position, range location of the word(s).

You can transfer part of Assigning match variables to user variables:

```
$$stuff = _0
```

but user variables only have a single piece of data. So on assignment you lose 2 of the 3 pieces from the match variable. You can choose which words (original or canonical) when you assign.

```
$$stuff = '_0 #  original words
```

You can store positional data onto a different variable using `^position(start _0)` or

```
^position(end _0).
_0 = $$stuff
```

When you assign onto a match variable from a user variable, you make both original and canonical values of the match variable the same, and the positional data is set to 0.

```
_0 = _10
```

This is a transfer from one match variable to another, so no data is lost.

One unusual property of match variables is that they are not cleared between volleys. This makes them the ONLY way you can pass data between volleys on a server where different users are involved.

Note: Match variables have a 20,000 character limit.

JSON dotted notation for variables

If a variable holds a JSON object value, you can directly set and get from fields of that object using dotted notation.

This can be a fixed static fieldname or a variable value- `$myvar.$myfield` is legal.

Dotted notation is cleaner and faster than `^jsonpath` and `jsonobjectinsert` and for get, has the advantage that it never fails, it only returns null if it can't find the field. On the other hand, assignment fails if the path does not contain a json object at some level.

```
$x = $$$obj.name.value.data.side
$$$obj.name.value.data.side = 7
```

Assigning `null` will remove a JSON key entirely. Assigning `"" ^""` will set the field to the JSON literal `null`.

Fact dotted notation for variables

If `$$f` holds a fact id, then

```
$$f.subject
$$f.verb
$$f.object
```

will return those components. You may NOT, however, assign into them.