ChatScript External Communications

© Bruce Wilcox, gowilcox@gmail.com brilligunderstanding.com

Revision 3/8/2015 cs45.2

ChatScript is fine for a chatbot that does not depend on the outer world. But if you need to control an avatar or grab information from sensors or the Internet, you need a way to communicate externally from ChatScript. There are three mechanisms: embedding ChatScript inside another main program which will do the machine-specific actions, calling programs on the OS from ChatScript where ChatScript remains in control, and getting services via the Interet from ChatScript.

Embedding ChatScript within another program

Most users use the executable versions of ChatScript for LINUX or Windows that are provided with a ChatScript release. Sometimes users have to build their own executable for the Mac. These are all building a complete ChatScript engine that is the main program.

Why would you want to embed ChatScript within another program – meaning that ChatScript is NOT the main program but some other code you write is? Typically it's to build a local application like a robot or a mobile chatting app. In this context, the main program is controlling the app, and invoking ChatScript for conversation or control guidance. You add ChatScripts files to your project and compile it under a C++ compiler. ChatScript is C++; it allows variables to be declared not at start, etc. So it won't compile under a pure C compiler. If you are trying to something more esotheric (dynamic link library or invoking from some other language) you need to know how to compile and call C++ code from whatever you are doing.

Embedding Step #1

First, you will need to modify common.h and compile the system. Find the // #define NOMAIN 1 and uncomment it. This will allow you to compile your program as the main program and ChatScript merely as a collection of routines to accompany it. Since this is an embedded program, you can also disable a bunch of code you won't need by uncommenting:

```
        // #define DISCARDSERVER 1
        // #define DISCARDCLIENT 1
        // #define DISCARDSCRIPTCOMPILER 1
        // #define DISCARDTESTING 1 – if your script will execute test functions then you must keep
```

Embedding Step #2

Second, you will need to invoke ChatScript from your program. This means the following:

Call InitSystem(...). It takes the typical argc/argv arguments and you will want to supply a bunch of them to specify how much memory to use (assuming you don't have a full Linux system you can just let rip). The InitSystem call also expects 3 filesystem paths (based on IOS requirements) though they can be the system. The first path is the path for files that never change. Typically this is the path leading to the DICT folder. The next path is for files that can be read, but possibly changed by download from the outside (e.g, LIVEDATA, TOPIC). The last path is for files the engine may write out (e.g., USER).

To call these routines, your code will need a predeclaration of these routines. Just copy the routine declarations from ChatScript and put at the start of your file, e.g.,

```
unsigned int InitSystem(int argc, char * argv[],char* unchangedPath,char* readonlyPath, char*
writablePath);
void InitStandalone();
void PerformChat(char* user, char* usee, char* incoming,char* ip,char* output);
```

You need to add all the CS .cpp files to your build list. Then you can compile your code along with ChatScripts.

Embedding Step #3

ChatScript is now ready to act. In the future you will call it with a message, and it will return with an answer that you do whatever with. The routine is:

```
void PerformChat(char* user, char* usee, char* incoming,char* ip,char* output)
```

The user string is the user's id. Since this is an embedded app, there will likely be only one user ever, so this can be hardcoded to anything you want. It will show up in the log file, and if you upload logs for later analysis, you will prefer this be unique in some way – a phone id or whatever.

IP is like the user string, a form of identification that appears in the log. It may be null or the null string, since the user id will probably be sufficient.

The usee is the name of the chatbot to talk with. This usually can be defaulted to the null string if you only have one bot in the system.

Incoming is the message from the user. The first time you start up a session, this should be a null string to inform the system a conversation is starting. Thereafter, you would just pass across the user input.

Output is the buffer where ChatScript puts out the response.

Alternate embedding Step #3

A variant of PerformChat is

    void PerformChatGivenTopic(char* user, char* usee, char* in,char* ip,char* out,char* topic)

The topic you give is presumed to be a block of memory into which you have written:

1. the null terminated name of the topic
2. all the rule of the topic in compiled form, eventually null terminated.
3. Followed by a block terminator of this string “`000 “ (ascii 3 zeros and a blank)`

The system adds this one topic into the topic system as though it were part of the system, which it is, but only for the duration of the call. This is useful if you want to synthesize a topic out of rules you have lying around. You get rules in proper format by using :topicdump topicname to generate rules one per line. You will then be responsible for concatenating some number of these rules int this memory block later. If a topic of this name already exists, the resident topic will be disabled during the call and restored at completion of the call. The block of memory topic will inherit the keywords and topic flags of the preexisting topic. Otherwise it has no keywords and no topic flags.

If you have no resident topic of this name you cannot have keywords on this topic, but you can create a topic in the normal system that has keywords and invokes this topic. And you can, as always, pass in and out variable values using out-of-band [] communication. Furthermore, the actual tag of the rule that matched is actually available to you at the end of the out buffer, just after the null terminated output.

Tracking User Data

Normally ChatScript tracks user data in a topic file in USERS. If you want to maintain your own state and not use CS tracking, then set the startup command line paramater to

```
cache=0x0
```

Memory Issues

Depending on what your platform is, you may need to reduce memory. The full dictionary, for example, may take 25MB and facts for it another chunk. For mobile apps for a price, I can build a mini-dictionary which is about 1/3 the size. Contact me if you need one.

The parameters I'd pass into most applications that are memory short, is to see what the used dict count is and make your "dict=nn" parameter be 1000 more. Same for "fact=nn". "Text=nn" should probably be 20. You might reduce the size of buffers from 80kb to 20kb if your bot doesn't say long stuff. And the bucket hash size should probably be around 10K.

Calling Outside Routines from ChatScript

ChatScript has several routines for calling out synchronously to the operating system.

^system(any number of arguments) - the arguments, separated by spaces, are passed as a text string to the operating system for execution as a command. The function always succeeds, returning the return code of the call. You can transfer data back and forth via files by using ^import and ^export of facts.

^popen(commandstring 'function) – The command string is a string to pass the os shell to execute. That will return output strings (some number of them) which will have any \r or \n changed to blanks and then the string stripped of leading and trailing blanks. The string is then wrapped in double quotes so it looks like a standard ChatScript single argument string, and sent to the declared function, which must be an output macro or system function name, preceded by a quote. The function can do whatever it wants. Any output it prints to the output buffer will be concatenated together to be the output from ChatScript. If you need a doublequote in the command string, use a backslash in front of each one. They will be removed prior to sending the command.

E.g.,

```
outputmacro: ^myfunc(^arg)
^arg \n
topic: ~test( testing )
u: () popen( "dir *.* /on" '^myfunc)
```

output this:

```
" Volume in drive C is OS"
" Volume Serial Number is 24CB-C5FC"
""
" Directory of C:ChatScript"
""
```

```
"06/15/2013 12:50 PM <DIR> ".
"06/15/2013 12:50 PM <DIR> .".
"12/30/2010 02:50 PM 5 authorizedIP.txt"
06/15/2013 12:19 PM 10,744 changes.txt"
"05/08/2013 03:29 PM <DIR> DICT"
.       .. ( additional lines omitted)
" 49 File(s) 29,813,641 bytes"
" 24 Dir(s) 566,354,685,952 bytes free"
```

'Function can be **null** if you are not needing to look at output.

^tcpopen(kind url data 'function) – analogous in spirit to popen. You name the kind of service (POST, GET), the url (not including http://) but including any subdirectory, the text string to send as data, and the quoted function in ChatScript you want to receive the answer. The answer will be read as strings of text (newlines separate and are stripped off with carriage returns) and each string is passed in turn to your function which takes a single argument (that text). :trace TRACE_TCP can be enabled to log what happens during the call.

'Function can be **null** if you are not needing to look at output. The system will set $$tcpopen_error with error information if this function fails.

When you look at a webpage you often see it's url looking like this:
http://xml.weather.com/weather/local/4f33?cc=*&unit="+vunit+"&dayf=7"

There are three components to it. The host: xml.weather.com. The service or directory: /weather/local/4f33. The arguments: everything AFTER the ?. The arguments are URL-encoded, so spaces have been replaced by +, special characters will be converted to %xx hex numbers. If there are multiple values, they will be separated by & and the left side of an = is the argument name and the right side is the value. When you call TCPOPEN, normally you provide the host and service as a single argument (everything to the left of ?) and the data as another argument (everything to the right of ?). Since ChatScript URL encodes, you don't. If you don't know the unencoded form of the data or you don't think CS will get it right, you can provide URL-encoded data yourself, in which case make your first argument either POSTU or GETU, meaning you are supplying url-encoded data so CS should not do anything to your arguments.

Below is sample code to find current conditions and temperature in san francisco *if you have an api key to the service.* It calls the service, gets back all the JSON formatted data from the request, and line by line passes it to ^myfunc. This, in turn, calls a topic to hunt selectively for fragments and save them, and when all the fragments we want have been found, ^myfunc outputs a message and stops further processing by calling ^END(RULE). Note that in this example there is no data to pass, everything is in the service named, so the data value is “”.

```
outputmacro: ^myfunc (^value)
       $$tmp = ^value
        nofail(RULE respond(~tempinfo))
        if ($$currentCondition AND $$currentTEMP)
         {
             print( It is $$currentCondition.  )
             print(The temperature is $$currentTemp. )
             ^END(RULE)
         }

topic: ~tempinfo system repeat keep()
u: (!$$currentCondition)
             $$start = findtext($$tmp $$pattern1  0)
             $$findtext_start = findtext($$tmp ^"\"" $$start)
             $$currentCondition = extract($$tmp $$start $$findtext_start )

u: ($$currentCondition)
             $$start = findtext($$tmp $$pattern2  0)
             $$findtext_start = findtext($$tmp ,  $$start)
             $$currentTemp = extract($$tmp  $$start $$findtext_start)

topic: ~INTRODUCTIONS repeat keep (~emogoodbye ~emohello ~emohowzit name )
t: ^keep() Ready. Type “weather” to see the data.

u: (weather)
       $$pattern1 = ^"\"weather\":\""
       $$pattern2 = ^"\"temp_f\":"
       if ( tcpopen(GET api.wunderground.com/api/yourkey/conditions/q/CA/San_Francisco.json ""
       '^myfunc)) { hi }
       else {$$tcpopen_error}

There is a subtlety in the ^myfunc code in that it uses ^print to put out the result. Just
writing:
 if ($$currentCondition AND $$currentTEMP)
         {
               It is $$currentCondition.
              The temperature is $$currentTemp.
              ^END(RULE)
         }
will not work, because that output is being generated by the call to ^tcpopen, which is in the
test part of the if, so everything it does is purely for effect of testing a condition. The
generated output is dicarded. If you moved the output generation to the { } of the if, things
would be fine. E.g.,
if ( tcpopen(GET api.wunderground.com/api/yourkey/conditions/q/CA/San_Francisco.json ""
'^myfunc)) {
                     It is $$currentCondition.
                     The temperature is $$currentTemp.
              }
       else {$$tcpopen_error}
```

Doing the output without using ^print is my preferred style; it is easier to see what is going on for output if it is not hidden deep inside some if test.

^export(name from) From must be a fact set to export. Name is the file to write them to. An optional 3rd argument “append” means to add to the file at the end, rather than recreate the file from scratch. Obviously, you must first have done something like ^query to populate the fact set. Eg.

```
  ^query(direct_sv item label ? -1 ? @3)
   ^export(myfacts.txt @3)
```

^import(name set erase transient) – Name is the file to read from. Set is where to put the read facts. Erase can be “erase” meaning delete the file after use or “keep” meaning leave the file alone. Transient can be “transient” meaning mark facts as temporary (to self erase at end of volley) or “permanent” meaning keep the facts as part of user data. Eg

```
    ^import(myfacts.txt @3)
```