ChatScript External Communications

© Bruce Wilcox, gowilcox@gmail.com

Revision 11/30/2013 cs3.74

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 two mechanisms, embedding and calling.

Embedding ChatScript within another program

Why would you want to embed ChatScript within another program? Typically its 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.

Gesture Input and Output

ChatScript can neither see nor act, but it can interact with systems that do. The convention is that out-of-band information occurs at the start of input or output, and is encased in [ ]. ChatScript does not attempt to postag and parse any input sentence which begins with [. It will automatically not try to spellcheck it also. In fact, the […] will be split off into its own sentence.

Gesture output needs to be first, which means probably delaying to one of the last things you do on the last sentence of the input, and using ^preprint(). E.g.

```
u: ($$outputgesture) ^preprint( \[  $$outputgesture \] )
```

You can hand author gestures directly on your outputs, but then you have to be certain you only output one sentence at a time from your chatbot (lest a gesture command get sandwiched between two output sentence). You also have to be willing to hand author the use of each gesture. I prefer to write patterns for common things (like shake head no or nod yes) and have the system automatically generate gestures during postprocessing on its own output.

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.

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"
```

^tcpopen(kind url data 'function) – analogous in spirit to popen. You name the kind of service (POST, GET, OTHER), 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. E.g., here is a call to parse german using an existing server (but to use this seriously you need your own mate-tools server because this one cannot handle any real load).

```
outputmacro: ^myfunc (^value)
       _0 = ^burst(^value " ")
       if (_1 != "returntype=rdf")
```

```
	{
		$$tmp = join(~ _4)
		$$loc = _0
		mark($$tmp $$loc )
	}

topic: ~germanparse ()

u: () tcpopen(POST "de.sempar.ims.uni-stuttgart.de/parse" "sentence=ich+bin&returnType=rdf"
'^myfunc)
```

^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.

^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.