ChatScript External Communications

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.

OOB Communication

ChatScript can actually run 2 distinct message channels at the same time, one with the user and one with the application. Communications with the application are called out-of-band (oob) communication and consists of placing such data inside of [ ] before the user's input or output.

```
[category: legal source: ios]  I have a problem. -- input oob + user
[video: thinking audio: humming] What kind of problem do you have? -- output oob + user
```

The format of things inside the leading [] in those messages is entirely up to you. The above examples show tagged pairs. They could equally be massive JSON data blobs.

The oob messaging conventions are directly supported by ChatScript. It will not apply NL processing (spell-correct, pos-tagging, parsing) to data within inbound oob. Additionally, normally inbound data is limited to 254 tokens per sentence (a volley may have any number of sentences). But if your oob data is a JSON structure, that is able to bypass the limitation (returning just a single token representing all the JSON).

OOB UI Conventions

While not required, many applications that embed ChatScript are well served by implementing the standard out-of-band (oob) communication mechanisms that CS supports on its webpage interface. The standard UI ones are:

```
[callback=1000]
[loopback=3000]
[alarm=10000]
```

Each message specifies a time in milliseconds. Callback says, if the user doesn't start typing within the time limit, automatically call ChatScript back, passing in as input:

`[callback]`

This request only applies a single volley and gives the chatbot control to keep going if the user doesn't start to reply. If the user starts a reply, the timer is cancelled and callback will not happen.

Loopback does the same thing, but automatically for every volley. So whenever CS sends output, a timer begins and if the user does not start replying within the time, the application calls CS with the message

`[loopback]`

If the user does start typing, it will not cause a loopback on the current output, but the next output starts the timer again.

Alarm says let the specified time elapse and then as soon as the application has control, call cs with the input

`[alarm]`

It is a one-shot alarm. Receiving a time of 0 cancels the corresponding timer, e.g., [loopback=0] or [alarm=0] or [callback=0]

Implementing these features require simple scripting in your bot. And then client-side creation of timers based on seeing the messages in outbound oob (eg in JavaScript webpage).

Calling Outside Routines from ChatScript

ChatScript has several routines for calling out synchronously to the operating system that are fully documented in the system functions manual. Here is just a brief survey of them.

`^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.

`^popen( commandstring 'function )`

The command string is a string to pass the os shell to execute. That will return output strings that will be sent to the declared function, which must be an output macro or system function name. The function can do whatever it wants.

`^jsonopen( kind url postdata header )`

this function queries a website and returns a JSON datastructure as facts.

^jsontree(name)

It prints out a tree of elements, one per line, where depth is represented as more deeply indented.

^jsonparse(string)

this parses into facts exactly as ^JSONOPEN would do, just not retrieving the string from the web.

^jsonpath(string id)

retrieves a datum from json-mapped facts, given the path to walk to get it.

^tcpopen(kind url data 'function)

analogous in spirit to popen but you are requesting data from the web and processing the returned output via your function.

‘^export(name from)

From must be a fact set to export. Name is the file to write them to.

^import(name set erase transient)

Name is the file to read from. Set is where to put the read facts.

WebSockets

CS is not a websocket server. It can, however, connect to some existing websocket endpoint and be a client for it.

^websocketinit(url) - connect to the given url

^websocketclose() - disconnect from opened socket

^websocketsend(message) - send message to opened socket

^websocketreceive(timeout) - listen for response back. timeout value of -1 means block until message received. 0 means retrieve immediately if something is there, otherwise return with nothing. Other values are milliseconds will be willing to wait for response.

If you use the param websocket=url the system on startup will become a websocket client and use that as its input/output loop channel. If you also supply parameter websocketmessage="send this first" then that message will be sent when the socket is opened.

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
```

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.

The extra considerations in a stand-alone-based app are three-fold: memory size, logging, and upgrading.

Things like iOS devices may restrict your app to somewhere in the 20MB size, and if you go beyond that you risk having your app killed by the OS at any moment. In that context, you cannot afford a full ChatScript dictionary and need a "miniaturized" one that occupies maybe 1/3 the normal space.

A large ChatScript memory footprint including the script for the bot is around 15-18MB. The miniaturized dictionary is not something you can easily build yourself and must be created by me (normally for a fee of $1K for any number of recreations of a mini-dictionary when you give me the source to build it from).

Log files are generated locally in the device and must somehow migrate to a server if you want analytics. Typically this is done when the user launches the

app and if an internet connection exists and it has been long enough since the last upload, the log files are uploaded to a server and cleared from the client.

Upgrade issues involve the desire to fix or augment ChatScript content without requiring the user download a new edition of the app. That is, on startup the app would check with the server and if specific files (eg the entire TOPIC folder) have changed, it downloads those changed files. It either then initializes ChatScript (the changes take place immediately) or if it has already initialized ChatScript the changes will take place starting with the next user startup of the app.

Embedding Step #1

First, you will need to modify common.h and compile the system. You need to add all the CS .cpp files to your build list. Or if you are running Windows, you can just use the ChatScriptDLL in the BINARIES folder.

Otherwise, 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

To embed CS within a client, you need to perform three calls. To call these routines, your code will need a predeclaration of these routines. The first is

```
InitSystem(int argc, char * argv[],char* unchangedPath, char* readablePath, char* writeableP
```

where you pass in various command line parameters to control things like logging or memory usage. Only the first two parameters are required, the remainder being optional and defaulted to NULL.

The three paths will normally be NULL, unless you are under iOS where their devices access files from different areas. All paths refer to being in the ChatScript directory, but different folders are stored under different paths based on expected use. The unchangedPath would be the read-only folders DICT/ and src/ and LIVEDATA, which never get changed. The readablePath would be the folder TOPIC and VERIFY and `authorizedIP.txt`, where the app doesn't change it but you might download replacement data. The writeablepath would be for USERS and TMP, which CS changes with every volley (if TMP exists).

ChatScript assumes its own directory is the current working directory when trying to access folders. If that is not true, you could pass the full path to the

CS directory as the arguments for each of the path arguments. The userfiles parameter allows you to pass a structure of routines that take over for all file reads and writes, so you can replace CS's access to files with your own mechanisms.

Normally for an embedded system you would not have a VERIFY folder and probably not a TMP folder unless you were supporting :retry. Nor would you have authorizedIP because you wouldn't be using debugging commands.

The second call is the actual workhorse...

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

or

```
int PerformChatGivenTopic(char* user, char* usee, char* incoming,char* ip,char* output,char*
```

PerformChat is told the user name, the bot name (usee), the incoming message, the ip address, and where to put the 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.

The GivenTopic form can be given an extra binary topic to add to the system, for dynamic topic generation (see below). Ip can be null. Usee can be the empty string "" which means use default bot. Incoming can be null string "" which means conversation is being initiated. Output is the buffer to put output into. Return value is the volleycount of this volley.

PerformChatGivenTopic takes data in the form output by :topicdump into `TMP/tmp.txt`. That can be read in by :extratopic as a test, which merely reads in all lines, removing the cr/lf and concatenating the data together into one long text string. This is passed thereafter into PerformChatGivenTopic. The data output by :topicdump includes topic flags and keywords, and new topics brought in can use these just like normal topics. For

```
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.

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.

The third call is when you want to release CS and shutdown...

```
void CloseSystem();
```

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.