

Introduction to BSW Command Puppet Programming:

By Jeff Bakalchuck

Jeff@theboomer.us

This introduction to Command Puppet Programming is divided into 5 sections. The first section is a general overview and is designed for people without prior programming experience. Section 2 covers basic programming and section 3 covers advanced programming. Section 4 covers style issues and hints and tips. The 5th and final section covers debugging of puppet.

Section 1: Command Puppet Overview:

BSW command puppets are computer programs that run while connected to the BSW server. The puppet program runs separately from your client session and can remain running after you log off. These puppets can interact with BSW players and issue commands. If you have some programming experience it will be fairly easy to learn the puppet programming language. The puppet language is very similar to the REXX programming language. For those of you with no prior programming experience, the learning process will be slightly more difficult. It is important to remember that you will make mistakes, all programmers do. It's part of the learning process. Also, one very important thing to remember is, when programming puppets case matters, so END is not the same as end. Examples you see that are all uppercase are that way for a reason. One of the most common errors is to forget to use uppercase.

So, what can a puppet do?

Well, puppets react to system events. For example, a puppet could be designed to display a message when someone enters the same room as the puppet. Puppets are designed to react to specific events. In the programming world, we call this "Event Driven Programming". In simple terms this means the puppet doesn't do anything until an event occurs. The puppet then jumps into action and executes the the instructions in its program. Programmers call these instructions "Code".

Before we get too deep into the details, let take a look at the code for a simple puppet:

```
PUPPET hiya
INFO: "hello puppet"
ACTION start
WHEN APPEAR * DO showsup
END
ACTION showsup
>> Hello my name is hiya, welcome to my room
END
PUPPETEND
```

The first line names the puppet, in this case "hiya". The 2nd line sets the puppet information line, so if anyone does a /info hiya command, that's what they will see in the info box. The line that reads "ACTION start" is where the fun begins. The start ACTION says what to do when the puppet begins. The next line defines an EVENT that the puppet will react to. We call this an event listener. In this example the event is APPEAR *. This means the puppet will "listen" for someone to appear in the room. The * means everyone.

We could code:

```
WHEN APPEAR username DO showsup
```

In that case, the puppet would only react to one person. The puppet then jumps to the ACTION showsup and executes that code. In our example we have:

```
>> Hello my name is hiya, welcome to my room
```

The >> tells the puppets to display some text in the main BSW window.

You'll notice that ACTION, WHEN, DO, END, APPEAR and other words are in uppercase. These are reserved words and they must be coded in uppercase letters. On the other hand, showsup is a programmer defined name and can be anything the programmer wishes, except a reserved word. It's a good idea to make your programmer defined words lowercase. This makes it easier to spot errors.

The last line simply marks the end of the puppet code.

Puppets are restricted in what they can do. They cannot kick, summon or beam players and they cannot play in any games.

Also remember puppets are event driven, so they just sit there waiting for an event, just as hiya sits and waits for someone to appear.

Now that you have seen a simple puppet lets move on to section 2 and basic command puppet programming

Section 2: Basic Command puppet programming:

When you write a puppet you create a plain text file that must be uploaded to a website. BSW does not provide any space to host your puppets. Since it's a plain text file, you can use notepad under windows or any other editor/word processor that can save a file as plain text. Once you have the file uploaded to a website, you can start it with the following command:

```
/puppet CP name http://url
```

So to start "hiya" you might type

Now that you know how to start a puppet, let look at a more complex puppet:

```
PUPPET counter  
INFO: "counting puppet"  
ACTION start  
EVAL count = 0  
WHEN APPEAR * DO showsup  
END  
ACTION showsup  
EVAL count = [count] + 1  
>> Hello my name is hiya, welcome to my room, you are person number [count] to visit me  
>> /tell xxx someone entered the room  
END  
PUPPETEND
```

This puppet has a few additions to "hiya". Line 4 reads "EVAL count = 0". In this puppet, count is a variable, it is used to store the number of people that have appeared in the room. The name of the variable is chosen by the programmer. The reason it is called a variable, is that its value can change over time. Some of the things you can do with variables are: Add, Subtract, Multiple, Divide, Display. You can also compare the value of a variable to other values. The EVAL statement allows you to set the value of a variable. Line 8 shows us how to add 1 to a variable. Notice the use of [count]. Whenever you enclose a variable inside [], you are saying use the value of that variable. In line 9, we use [count] in a display command. If you did not use the [] you would get the word count, not the value of count. Lastly Line 10 shows an example of a puppet executing a command. In this case a /tell command, which sends a private message to xxx. One important note, when executing / commands you must have exactly one space between the >> and the /.

Before we get into all the nifty commands, lets look at one more simple example:

```
PUPPET greeter
INFO: "greeting puppet"
ACTION start
EVAL count = 0
WHEN APPEAR * DO showsup
WHEN KEYWORD "hello" FROM * DO howdy
END
ACTION showsup
EVAL count = [count] + 1
>> Hello [WHO] , welcome to my room
END
ACTION howdy
>> /tell [WHO] well hello to you too
IF [WHO] == TheBoomer DO
>> /tell TheBoomer Hello Boomer, I hope you are having a nice day.
PUPPETEND
```

In this puppet, we've added a few new features. First we have a 2nd WHEN statement, so this puppet will "listen" for 2 different events. In this example, if anyone says the word hello, the puppet will execute the ACTION howdy. Also, notice we are using a new variable named WHO. WHO is a system variable, which means it is set automatically by the system, we do not need to use an EVAL statement. When a KEYWORD or APPEAR event occurs, the system automatically places the name of the person associated with that event in the variable WHO. So now our puppet knows who said hello. Lastly, we have an IF statement. The IF statement tests for a specific condition. In this case, if the value of WHO is TheBoomer. If the condition is true, the following line of code is executed. IF statements can be very complex, but allow you to write puppets that do a wide range of actions.

If you have followed these 3 examples, then all you need to know is the list of possible puppet commands, lets start with the basic event listeners:

WHEN KEYWORD x DO action

Example: WHEN KEYWORD "I hate lag" DO meto

WHEN KEYWORD x FROM y DO action

The FROM is optional and allows you to listen for keywords from specific users

Variables:

WHO = username of person that typed the keyword

ROOM = room that that person is in

TYPE = CHAT, TELL, CTELL, GTELL method used to type the keyword

IGNORE KEYWORD x FROM y

This allows you to ignore keywords from certain users. y can be a list of names, we will cover lists in the next section.

IGNOREALL KEYWORD FROM y

Ignores any keywords from a specific user

WHEN MATCH string FROM y DO action

Example: WHEN MATCH *no* DO action1

The * means any text including spaces

WHO,ROOM, TYPE = same as for WHEN KEYWORD

SUBST1 = the text from the first *

SUBST2 = the text from the second *

SUBSTn = the text from the nth *

So, if someone typed:

Hello Mr. Benno how was you day?

SUBST1 = Hello Mr. Ben

SUBST2 = how was you day?

IGNORE MATCH string FROM y

IGNOREALL MATCH FROM Y

These are similiar to IGNORE KEYWORD and IGNOREALL KEYWORD

WHEN APPEAR list DO action

WHO = username of person that appeared,

Example: WHEN APPEARS "moe larry curly" DO wiseguys

*WHEN APPEAR * DO action*

use * to mean all usernames

WHEN APPEAR username DO action

IGNORE APPEAR list

IGNOREALL APPEAR

WHEN DISAPPEAR list DO action

IGNORE DISAPPEAR list

IGNOREALL DISAPPEAR

WHEN NEWROOM DO action

ROOM = room number of the new room

This event occurs when the puppet is summoned or beamed to a new room.

WHEN TIME hh:mm DO action

At a specific time perform the action, mm is optional for times on the hour. Time is BSW server time.

WHEN TIMER x DO action

x is some amount of seconds, so WHEN TIMER 3600 would mean once per hour.

This event performs the action once every x seconds, over and over, as long as the puppet is running.

WHEN PING FROM list DO action

FROM is optional

WHO = username that pinged the puppet

WHEN KICKED DO action

WHO = username of person that kicked the puppet.

Now lets look at a few system commands:

GETDATE

places the current date into the following variables

HOUR,MIN,SEC,DAY.MONTH,YEAR,DAYOFWEEK note: sunday=1, monday=2 etc.

GETINFO username

returns information about the username in the following variables:

CITY, CITYNR, LANGUAGE, PUPPET(TRUE OR FALSE), PLAYING(TRUE OR FALSE), TITEL, RANK,
SEX(n,m,w),TUTOR(TRUE OR FALSE),REPORTER(TRUE OR FALSE), AMT(city officer title)

GETWHO

WHO = list of all the people in the room

GETROOMINFO

ROOM = current room number

NAME = name of room

GAME = name of game in the current room

WHEREIS PUPPET/OWNER/STARTER

ROOM = room number where puppet or owner or starter is

WHOIS PUPPET/OWNER/STARTER

WHO = name of puppet or owner or starter

SET variable value

UNSET variable

sets a variable to a value or removes value from a variable

EVAL variable = value or equation

can be used to set a variable to a calculated value

SLEEP x

x is some number of seconds, puppets pauses and does nothing, this is useful for slowly down output lines

In addition to the event listeners and the system commands, puppets can execute any of the following user or / commands:

/who
/room
/tell
/mtell
/reset
/shout
/name
/hook
/channel
/gtell
/ctell
/leave
/game
/blacklist
/inhabitants
/cityinfo
/mutectell

For details on exactly how these commands work, see the help files on the BSW website.

The last piece of the basic programming language are DO and IF statements.

The DO statement simply says perform an action. We use this to simplify the code by breaking it up into smaller pieces. Like the following example.

```
ACTION longcomplexcode  
DO part1  
DO part2  
DO part3  
END
```

IF statements can be the the most difficult part of your code to get working right, pay close attention to the code example, make sure you have uppercase and [] where necessary.

The basic format of the IF statement is:

```
IF condition
    statement 1
ELSE
    statement 2
```

If the condition is true then statement 1 is executed, if the condition is false then statement 2 is executed. The ELSE statement 2 is optional. It is very important to remember that statement 1 and statement can only be a single statement. If you require more than one statement, then you need to enclose the statements in a BEGIN/END block as such:

```
IF condition
BEGIN
    statement 1
    statement 2
    statement 3
END
ELSE
BEGIN
    statement 4
    statement 5
END
END
```

The types of conditions can be very complex, and in the next section we will cover them in detail, for now lets look at the 2 most common, testing the value of a variable equal to or not equal to a specific value.

```
IF var == value
```

```
IF var != value
```

Please note the double =, this is correct. When you compare 2 items in an if statement you must use a double =.

Lets look at another puppet to see how we can use IF statements.

```

PUPPET match
INFO: "if statement demo puppet"
ACTION start
WHEN APPEARS * DO showsup
END
ACTION showsup
IF [WHO] == HeidiKlum DO
BEGIN
>> Hey good looking
>> /tell TheBoomer Hey Boomer, your date is here.
END
ELSE
    BEGIN
    IF [WHO] == JanetReno
    >> /tell Ravnwyng someone here to see you
    IF [WHO] == curly
    DO nyuknyuknyuk
    END
END # this end is for the action showsup
ACTION nyuknyuknyuk
>> Hey Curly, Moe said to make a note that he's gonna kill you.
END # this end is for the action nyuknyuknyuk
PUPPETEND

```

In this example we indented the BEGIN/END block inside the ELSE. This is purely for readability of the code. Also notice the last 2 END statements. The # means, everything after this is a comment. Comments are not performed by the puppet, but rather are there to help people reading the puppet code to understand it.

Section 3: Advanced Command Puppet Programming.

The puppet programming language has a great many features. In the previous section we just touched on the features necessary to get started. In this section we will cover the remaining features. Of course the examples will get more complex also. At the beginning of a puppet is a series of settings lines. We've already seen one of these the INFO line. Now lets look at the rest:

LOGIN: string # message displayed when puppet starts

LOGOUT: string # message displayed when puppet ends

APPEAR: string # message displayed when puppet enters a room

DISAPPEAR: string # message displayed when puppet leaves a room

CITYCHAT: YES/NO

CASESENSITIV: YES/NO

SAVE: phrase

DEBUG: string

If you code CITYCHAT: YES, then the puppet will be able to see keywords entered in the city chat, and the puppet will also be able to issue /ctell commands. The default is NO.

If you code CASESENSITIV: YES then any keywords entered must match exactly the case as coded in the puppet. The default is NO.

The SAVE: phrase allows you to save data when the puppets ends and be able to use that data the next time the puppet starts. You get 100 variables, named SAVE1 thru SAVE100 that you can save. We'll see an example of this shortly. Please note that if the server is restarted, all data is lost. The phrase can be anything you wish, but should be unique across all of BSW, so the best choice is the same as the puppet name.

We'll cover DEBUG in section 5.

In addition to events, puppets can be designed to accept commands from users. This is still event driven programming, but in this case the event is a specific request from a user. The way we do this is with @ commands. To use an @ command, you define it the command immediately after the settings and before the ACTION start. The format is as follows

@command: action

Now, lets look at some code that uses these features:

PUPPET Archie

LOGIN: " Ohhhhhhhhh Archie"

LOGOUT: " That's all folks"

APPEAR: "Archie 2.1C is here"

DISAPPEAR: " See Ya Later."

INFO: "EnglishTown Information server. Version 2.1C Type @help for help information"

CITYCHAT: YES

SAVE: "Archie"

@news: news

ACTION start

>> /tell TheBoomer starting [SAVE98] [SAVE99]

IF [SAVE98] == 1

BEGIN

>> /tell TheBoomer I was kicked by [SAVE99]

SET SAVE98 0

SET SAVE99 NULL

END

>> /room C10

DO intro

WHEN KICKED DO Kicked

END

ACTION Kicked

SET SAVE98 1

SET SAVE99 [WHO]

IF [WHO] != TheBoomer

BEGIN

>> /tell [WHO] Please don't kick me, I'm a friendly puppet and I'm here to help people

>> /tell TheBoomer [WHO] just kicked me

END

HARAKIRI

END

ACTION intro

GETINFO [WHO]

>> Hello, My name is Archie, I can help you get information about EnglishTown

>> Type @help or @commands for information and how to use me

>> Thanks for visiting EnglishTown

END

ACTION news

GETINFO [WHO]

>> /tell [WHO] -----

>> /tell [WHO] Archie messaging now available

>> /tell [WHO] @msghelp for full details

>> /tell [WHO] -----

END

PUPPETEND

In this example we have defined @news to perform the ACTION news, so if anyone enters @news that action will get performed. We also have a DO statement in the start ACTION, this is just to break the code up and make it easier to read. The interesting part is what happens if someone kicks the puppet. We have a WHEN KICKED event listener, so if someone kicks the puppet, the ACTION kicked is performed. It is important to keep the amount of code in the kicked ACTION to a minimum. This is because there is a limited amount of time for the puppet to execute code before the system terminates it. However there is enough time for us to save the name of the person that kicked the puppet ☺ and send them a message. In this example, when someone kicks the puppet, we move a 1 to SAVE98 and the name of the kicker to SAVE99, then we send them a message and I also send a message to myself. HARAKIRI is a puppet command to terminate, it isn't required, but is helpful to ensure that data is saved.

Now look at the start ACTION. After sending myself a message, the next thing is we test SAVE98 to see if the puppet was kicked, and if it was, then it sends me a message saying who kicked the puppet. The puppet then is moved to room C10, the ACTION intro is then performed.

Notice that these data are saved even after the puppet terminated. That's the whole reason for SAVE1 thru SAVE100. You must set the setting SAVE: at the beginning of the puppet for the data to be saved.

Lets talk about Lists now. Remember the WHEN APPEARS event? In that event we could provide a list of usernames. In the puppet language a list is simply a series of items separated by a space. So the following could be used to initialize a list:

```
EVAL mylist = "abc def ghi xyz"
```

Lists can be used in many different ways. One of simplest ways is to use ELEMENTOF.

```
EVAL x = 1 ELEMENTOF mylist
```

```
>> x
```

This code would display abc. You could use the value of a variable in the place of 1, as long as the value of the variable is an integer.

```
EVAL x = [t] ELEMENTOF mylist
```

So what's so special about a list? Well say we have an action defined and we want the user to give us multiple pieces of data with that action, lists are a perfect way to process those multiple pieces. Let looks at some code from my puppet Scarne.

```
@rolls: rolls  
  
ACTION rolls  
IF EXISTS PARAM AND LISTLENGTH [PARAM] == 3 # There must be three parameters  
BEGIN  
FOR t IN "1 2 3" DO  
EVAL n[t] = [t] ELEMENTOF [PARAM] # number1 gets the first parameter and so on  
DO main  
END  
ELSE  
BEGIN  
>> Sorry [WHO], you gotta give Scarne all 3 numbers, ya know I ain't The Amazing Kreskin  
>> like: @rolls 5 6 7  
END  
END
```

Scarne was designed to give someone to percentage of a successful roll in the game can't stop, if they give Scarne their 3 climbing numbers. Therefore, Scarne requires 3 pieces of information. When a user types in an @ command, the system automatically takes any other data from that command and places in a list named PARAM. The beauty of using a list, is we don't know ahead of time, how many elements will be in the list. Using a list gives us flexibility. Let's look at the code.

The first thing we do, is test to see if PARAM exists and if the length of its list is equal to 3. If it isn't then we display an error message. If it is, now we must process the list. To do this we are using something known as a FOR loop. In a FOR loop, we execute the code multiple times, varying the value of a variable each time we execute.

The format is:

```
FOR variable IN list DO
```

In the case of Scarne, we set T to 1, then execute the EVAL and DO main, then we set T to 2 and again execute the EVAL and DO main, finally we set T to 3 and execute EVAL and DO main. Imagine if Scarne needed 20 params, the code would require very little modification

```
FOR t IN "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 18 19 20"
```

Now, notice something tricky in the EVAL statement? We have n[t], remembering the [t] means the value of t. this gives us variable n1 with the first param, if we need to display the value we could code [n[t]] or [n1].

In a FOR loop, the system automatically assigns the next value from the list, each time through the loop. The list doesn't need to be in order, it doesn't even need to be integers.

Another type of loop is a WHILE loop. With a WHILE loop we execute as long as a condition is true.

NOTE: WHILE loops can be very dangerous, if the condition never becomes false, the loop will run forever, or until you kick the puppet or the server is shut down. Be careful with them.

```
WHILE condition DO
```

```
BEGIN
```

```
Code
```

```
END
```

Let's look at an example of a WHILE loop from Archie.

```
ACTION gh
EVAL found = 0
EVAL i = 1
WHILE [i] <= 36 DO
BEGIN
EVAL j = [i] ELEMENTOF [guildlist]
IF [j] == [game] DO
BEGIN
EVAL found = 1
EVAL k = [j] ELEMENTOF [roomlist]
EVAL i = 37
>> /tell [WHO] The guildhall for [game] is [k]
END
EVAL i = [i] + 1
END
IF [found] == 0 DO
BEGIN
>> /tell [WHO] Sorry I didn't recognize the game you entered
>> /tell [WHO] Please use one of the following
>> /tell [WHO] [guildlist]
END
END
```

One of Archie's features is the ability to tell you the location of a GuildHall. Archie has a list of the 36 games on BSW and a list of the guild hall locations.

In this loop, the condition is $[i] \leq 36$. Before the loop i is set to 1. We then compare the game to the i th element of the list. If it matches we tell the user the location and set i to 37. The reason we set i to 37 is to stop the loop. We also set $found$ to 1, so when the loop ends, we can see if we found the guild hall or not.

There are many more possible ways to code a condition. Lets look at them.

condition 1 or condition 2

condition 1 and condition 2

NOT condition

ISEMPTY variable

EXISTS variable

ISINTEGER variable

string == string

string != string

variable < variable

variable > variable

variable <= variable

variable >= variable

string IN string

string INLIST list

string STARTSWITH string

string ENDSWITH string

We also have several methods to access a list:

FIRSTOF list

LASTOF list

FIRSTCHAROF list

LASTCHAROF list

LOWERCASE string

UPPERCASE string

LENGHTH string

LISTLENGTH list

x ELEMENTOF list

x CHAROF string

We can also do basic arithmetic:

int + int

int - int

*int * int (multiplication)*

int / int

int % int (remainder)

- int (negative)

Section 4: Style and puppet limits.

Server limits and restrictions:

Never start a command puppet within the first 10 minutes after the BSW server is restarted.

Maximum lines of code: 20,000

Maximum number of events: 1000

Maximum number of variables: 5000

Maximum number of @ commands: 500

Style issues:

- You should always have an @list command in your puppet. This command should generate a message to the user, via /tell explaining the purpose of the puppet and giving your username as a contact. This way people will know who's Puppet it is. Remember, a puppet could have a bug in it, that causes problems for other people.
- Use lowercase for all programmer named items. If you want to use multiple word named, you could capitalize the first letter of the 2nd word. So you could have a variable named testCase.
- Use blank lines to make the puppet code easier to read.
- The more comments the better.
- Break long sections of code into separate actions.
- Avoid very long variable names.
- Indent nested IF statements.
- Never have 2 programmer named items the same other than case, so don't use names and Names in the same puppet.
- Keep messages displayed to a minimum, use /tell wherever practical.

Section 5. Debugging puppets.

TO BE ADDED LATER