Wizard Script Language Manual

Manual Written by Keith Ledbetter Last Modified 8-Jul-01

How the Scripting Engine Works

The Wizard scripting engine is comprised of two separate pieces, although to you they will appear to be mostly one and the same. Part one is the script compiler, which compiles the English-like instructions from your scripts into a much more efficient tokenized module. This compilation takes place automatically whenever you execute a script that (a) has no associated object module or (b) has changed since it was last compiled. The second piece is the "script execution engine", which is what actually executes your script commands. The object module, when created, will have the same name as the original script with ".o" appended to its name (for example, "Drop.Cmd" will be "Drop.Cmd.o". This also allows you to distribute the compiled scripts that you've written without having to send your source code along with them.

The Wizard's script compiler engine is based on a standard recursive-descent parser algorithm that is used in many C compilers. This allows very flexible and powerful scripting capabilities with a grammar that can be easily modified to match the level of the script author. Since the compiler engine is fully recursive, conditional and loop control statements can be nested indefinitely (or until the memory pool dries up).

The compiler's parser routine is also very intelligent, giving the script author full control over their coding style. To illustrate this, look at the following three examples, which all pass the parser's grammar rules:

```
for x = 1 to 10 begin
      echo "Hello there!"
end
```

or:

```
for x=1 to 10
      begin
            echo "Hello there!"
      end
```

or even:

```
for
  x = 1
   to
     10 begin
           echo "Hello there!"
        end
```

Granted, the last example above takes "script readability" to an all-time low; however, it does illustrate the "free form" parsing that the script compiler's engine will allow.

The following pages will describe the syntax of the Wizard Scripting language. These terms are used in defining the scripting language grammar:

variable is a variable name that must begin with an alphabetic character. Variable names may be up to 32 characters in length.

constant is a hard-coded value of either a double quoted string ("this is a string") or a simple number (1024).

Note: if you need to place a double-quote in a constant string, you must prefix a '****' character to the double-quote. For example,

"A string with the word \"quoted\" enclosed in quotes."

would evaluate to:

A string with the word "quoted" enclosed in quotes.

value is either a variable or a constant value.

{statement} is used to show either a single statement or a block of statements. Blocks are started by the keyword "**Begin**" and terminated by the keyword "**End**". You may also use the C-style shortcuts of '**{**' and '**}**' to mark a code block. In other words, these are identical:

```
if ( a = b )
  begin
      ..do this..
      ..do that..
  end

if ( a = b )
{
  .. do this..
  .. do that..
}
```

operator denotes one of the following arithmetic operators:

=	(equal to)
<	(less than)
>	(greater than)
!=	(not equal to)
!<	(not less than)
!>	(not greater than)
==	a special string operator that checks for a string within another string. In other words:

```
if ( "abcdefg" = "cde" )
```

would evaluate to FALSE, since the two strings do

not equal (the single '=' means they must match exactly)

```
if ( "abcdefg" == "cde" )
```

would evaluate to TRUE, since "cde" does occur in the first string.

!== Substring **not** found in string.

condition is one (or more) conditional expressions in the following format that resolve to either true or false:

value operator value

Multiple conditions are separated by the words **AND** and **OR**. You can also use the C-style shortcuts of **&&** (and) or **||** (or).

[...] denotes optional parameters.

a | b The vertical bar denotes either/or settings (i.e.: either a or b)

%variable The script engine supports environment variables that give you access to much of the data on your character. These variables are always preceeded by a % sign, and are as follows:

Name	Format	Description
%0	string	for command line scripts, %0 is the entire command line the user entered.
%1..%9	string	for command line scripts, these are the 1st through 9th arguments the user specified on the command line.
%leftHand	string	the contents of your left hand
%rightHand	string	the contents of your right hand
%weapon %shield %container %sheath	string	These give you access to your current inventory environment variables. You set these values manually through the **Edit/Settings/Inventory** screen. You can also use these values in any keyboard macros that you write.

%directions string This string tells you the directions that are available in the room that you are currently in. This string will contain a mixture of the letters A - K, which signify the following available exits:

A	north
B	northEast
C	east
D	southEast
E	south
F	southWest
G	west
H	northWest
I	up
J	down
K	out

So, if %directions = "CFK", this means there are exits East, SouthWest, and Out.

%health integer Your current hit points.

%fatigue integer Your current fatigue.

%psi integer Your current psi points.

%web integer Your current web points.

%weight integer Your current weight points.

%gold integer Your current gold.

%silver integer Your current silver.

%status string This variable will contain words describing various aspects of your character. It is implemented as a string variable with spaces in between the various flag words. This means you should use the format

```
if %status == "<word flag>"
```

so that the partial string match feature will work.

The possible words in this variable are as follows:

Dead	Webbed
Kneeling	Invisible
Sitting	Intoxicated
Lying	Immobile
Standing	Unconcious
Stunned	Hidden
Bleeding	Joined
Calmed	Staunched
Diseased	Poisoned

For example, if your character was currently kneeling down, bleeding, and following another player, the %status string would be equal to:

"Kneeling Bleeding Joined"

Commands by Function Category

Variable Definition:

```
String      variable  [ , variable, variable ] [;]

Integer     variable  [ , variable, variable ] [;]
```

Variable Manipulation:

```
Set         variable = value  [ + value + value  ] [;]

Add         value to variable [;]

Subtract    value from variable [;]

Multiply    variable by value [;]

Divide      variable by value [;]
```

Displaying Messages to the Game Screen:

```
Echo        value  [ + value + value ] [;]
```

Processing Responses From the User:

```
Ask         string_variable, value  [ + value + value ] [;]
```

Communicating and Processing Responses From the Game:

```
Put         value  [ + value + value ] [;]
Get         variable [;]
Wait        [;]
NextRoom    [;]
WaitFor     value  [ + value + value ] [;]
Move        value  [ + value + value ] [;]
Pause       value [;]
```

String Parsing Routines:

```
GetWord           value, value, string_variable [;]
GetNumber         value, value, integer_variable [;]
GetWordAfter      value, value, string_variable [;]
GetWordsAfter     value, value, string_variable [;]
GetNumberAfter    value, value, string_variable [;]
```

```
StringToInteger   value, variable [;]
RemovePunctuation string_variable [;]
SubString         string_value, value, value, string_variable [;]
```

Parameter Manipulation:

```
Shift [;]
```

Conditional Execution:

```
If condition {statement}

Else {statement}
```

Loop Control:

```
For variable = value to value [step value] {statement}

While condition {statement}

Do {statement} While condition

Break

Continue
```

Unconditional Movement:

```
:LabelName

goto LabelName

call SubroutineName

return

ExitScript (or just Exit)
```

Miscellaneous:

```
Beep

Random

/* ... */   comments a block of lines

//          comments the remainder of a line
```

Commands and Their Use

Command: **String**

Syntax: string variable_name, variable_name2, ...

Use: All variables must be defined before they are used. The string verb allocates a variable in the global variable table that will be able to hold any string value up to 1024 bytes in length.

Example:

```
//    define 4 name variables that we'll use later on..
//    ------------------------------------------------

String      firstName, middleName;
String      lastName;
String      fullName;
```

Command: **Integer**

Syntax: integer variable_name, variable_name2, ...

Use: All variables must be defined before they are used. The integer verb allocates a variable in the global variable table that will be able to hold any numeric value between the values of -1,247,483,648 to +1,247,483,647.

Example:

```
//    define 2 numeric variables that we'll use later on..
//    ---------------------------------------------------

Integer     LastReturnCode, LoopCounter;
```

Command: **Set**

Syntax: set variable_name = value + value + value;

Use: The set command allows you to change the contents of either a string or a numeric variable that you have previously defined.

Note: The script compiler will not allow you to mix variable types. In other words, if you have defined a variable as an integer type, you will get an error message if you try to store a string value into that variable.

Example:

```
//    do some variable manipulation..
//    -------------------------------

String  firstName, middleName;
String  lastName, fullName;
Integer  LastReturnCode, LoopCounter;

Set firstName  = "John"
Set middleName = "Fitzgerald"
Set lastName   = "Kennedy"
Set fullName   = firstName + " " + middleName + " " + lastName;
Set LastReturnCode = 0;
Set LoopCounter = -50;
```

Command: **Add**

Syntax: add value to variable_name

Use: The add command is used to perform computations on an integer variable.

Note: The script compiler will not allow you to mix variable types. In other words, if you have defined a variable as a string type, you will get an error message if you try to add a value to that variable.

Example:

```
//    An ADD example...
//    -----------------

Integer Counter

Set Counter = 0

Add 5 to Counter              // counter is now = 5
Add 100 to Counter            // counter is now = 105
Add Counter to Counter        // counter is now = 210
```

Command: **Subtract**

Syntax: subtract value from variable_name

Use: The subtract command is used to perform computations on an integer variable.

Note: The script compiler will not allow you to mix variable types. In other words, if you have defined a variable as a string type, you will get an error message if you try to subtract a value from that variable.

Example:

```
//    A SUBTRACT example...
//    --------------------

Integer   Counter;

Set Counter = 25000
Subtract 100 from Counter          // counter is now = 24900
Subtract Counter from Counter      // counter is now = 0
```

Command: **Multiply**

Syntax: multiply variable_name by value

Use: The multiply command is used to perform computations on an integer variable.

Note: The script compiler will not allow you to mix variable types. In other words, if you have defined a variable as a string type, you will get an error message if you try to multiply using that variable.

Example:

```
//    A MULTIPLY example...
//    --------------------

Integer Counter

Set Counter = 10
Multiply Counter by 100       // counter now = 1,000
Multiply Counter by Counter   // counter now = 1,000,000
```

Command: **Divide**

Syntax: divide variable_name by value

Use: The divide command is used to perform computations on an integer variable.

Note: The script compiler will not allow you to mix variable types. In other words, if you have defined a variable as a string type, you will get an error message if you try to divide using that variable.

Example:

```
      //    A DIVIDE example...
      //    -------------------

      Integer   Counter

      Set  Counter = 25000
      Divide Counter by 10                // counter now = 2,500
      Divide Counter by Counter           // counter now = 1
```

Command: **Echo**

Syntax: echo value [+ value + value...][;]

Use: The echo command is used to print messages from your script to the Game's terminal window.

Example:

```
//    Using ECHO to display messages..
//    -------------------------------

String      str;
Integer     int;

set str = "Hello there, Wizard scripter!";
set int = 14257;

echo "Str is now <" + str + "> and Int is now <" + int + ">";

//    the above will print:
//
//    "Str is now <Hello there, Wizard scripter!> and Int is now <14257>"
```

Command: **If / Else**

Syntax: if condition [then] {statement} else {statement}

Use: The If command allows you to take actions based upon whether the condition specified evaluates to true or false. If statements can be nested to any level.

Note: If you want to perform more than one statement based on a condition, you **must** enclose the statements within a Begin/End block!

Note 2: ***Do not*** place a ";" terminator after the **else** statement, since a line only containing a ";" **is** a valid "statement" (unless, of course, that's what you really intend to do). See example 2 below.

Example 1:

```
//    if our psi falls below 20, get out!
//    -----------------------------------

if ( %psi < 20 )
{
   echo "Exiting -- psi is low!";
   exitScript;
}
else
{
   echo "Continuing prep spells...";
   ...do whatever...
}
```

Example 2:

```
//    this IF statement WILL NOT perform as expected,
//    because "else;" will parse out to "else <do nothing>".
//    ------------------------------------------------------

if ( %psi < 20 )
{
   echo "Exiting -- psi is low!";
   exitScript;
}
else;                                // this effectively says "do nothing"
{
   echo "Continuing prep spells...";  // this command block will always
   ...do whatever...                  // get executed, since it's not tied
}                                     // to the ELSE statement in any way.
```

Command: **For**

Syntax: for variable_name = startValue to endValue [step value]

Use: The For command is used to execute a statement or a block of statements for a predetermined number of times. The variable specified (the "counter" variable) will be initialized to "startValue" at the beginning of the loop and then incremented by "stepValue" each time through the loop (or by one if no "stepValue" is specified). The looping will continue until the variable is greater than "endValue".

Note: If you want to perform more than one statement with a for loop, you must enclose the statements within a Begin/End block!

Example 1:

```
//    beep the speaker 6 times to get their attention,
//    then put out an error message.
//    ------------------------------------------------

Integer  counter

if ( error != 0 )
{
   for counter = 1 to 3
   {
      beep;
      beep;
   }
   echo "An error of " + error + " occurred!";
   exitScript;
}
```

Example 2:

```
//    echo all even numbers between 0 and 20
//    --------------------------------------

Integer  counter

for counter = 0 to 20 step 2
   echo counter
```

Command: **Do**

Syntax: do {statement} while condition

Use: The Do command is used to execute a statement or a block of statements while the specified condition evaluates to true. Note that this differs from the While statement in that the {statement} block will always be executed at least one time before the condition is evaluated. Also note that there is **no** automatic incrementing of a loop counter done for you. You must be sure that you do not get into an infinite loop by writing code that will never satisfy the condition!

Note: if you want to perform more than one statement with a do/while loop, you must enclose the statements within a Begin/End block!

Example:

```
//    beep the speaker 6 times to get their attention,
//    then put out an error message.
//    ------------------------------------------------

Integer  counter

if ( error != 0 )
{
   set counter = 0;
   do
   {
      beep;
      beep;
      add 1 to counter;
   } while ( counter < 3 )
   echo "An error of " + error + " occurred!";
   exitScript;
}
```

Command: **While**

Syntax: while condition {statement}

Use: The While command is used to execute a statement or a block of statements while the specified condition evaluates to true. Note that this differs from the For statement in that there is **no** automatic incrementing of a loop counter done for you. You must be sure that you do not get into an infinite loop by writing code that will never satisfy the condition!

Note: if you want to perform more than one statement with a while loop, you must enclose the statements within a Begin/End block!

Example:

```
//    beep the speaker 6 times to get their attention,
//    then put out an error message.
//    ----------------------------------------------

Integer  counter

if ( error != 0 )
{
   set counter = 0;
   while ( counter < 3 )
   {
      beep;
      beep;
      add 1 to counter;
   }
   echo "An error of " + error + " occurred!";
   exitScript;
}
```

Command: **Break**

Syntax: break

Use: The Break command is used to break out of a for or a while loop. Execution will continue at the statement immediately after the scope of the current loop.

Example:

```
//    Tell the FOR loop to beep the speaker 1000 times,
//    but really only do it 5 times (don't ask me why..)
//    ------------------------------------------------

Integer  counter

if ( error != 0 )
{
   for counter = 1 to 1000
   {
      beep;
      if ( counter > 5 )
         break;
   }
   Echo "We'll come right here after the BREAK statement";
}
```

Command: **Continue**

Syntax: continue

Use: The Continue command is used to immediately return to the top of a for or while loop, thereby bypassing any code after the continue statement. Execution will continue at the "condition" check at the top of the loop.

Example:

```
//    Beep the speaker 10 times, but NOT if it's
//    the fifth or eighth time through the loop.
//    ------------------------------------------

Integer  counter

for counter = 1 to 10
{
   if ( counter = 5 or counter = 8 )
      continue;
   beep;
}
```

Command: **Goto**

Syntax: goto label_Name

Use: The Goto command is used to immediately pass execution to the statement following the specified label. A label must contain a colon in the first position of its name.

Note: Doing a GOTO command from within any nested block statement will be handled correctly.

Example:

```
//    Goto examples..
//    ---------------

if ( lastError = -43 )
      goto FileNotFound;
else if ( lastError = -120 )
      goto FolderNotFound;


... more code...


:FileNotFound
      echo "Source file doesn't exist!"
      ExitScript

:FolderNotFound
      echo "Source folder doesn't exist!"
      ExitScript
```

Command: **Call**

Syntax: call subroutine_Name

Use: The Call command is used to transfer execution to the subroutine named by the specified label. After the subroutine exits, execution will continue at the statement following the CALL command.

Example:

```
//    Call examples
//    --------------

if ( %1 = "" )                // did they specify a parameter?
{
   call GetParameters;        // nope, call a subroutine to get them.
   echo "Back from call";     // we'll return right here
}

...more code...

exitScript;                   // you MUST have an exitScript before any
                              // subroutine definitions, or you'll "fall
                              // thru" to them..not what we really want

subroutine GetParameters      // subroutines MUST be enclosed in Begin/End
{                             // ( or the 'C' shortcut of {} )
   echo "";
   echo "Getting parameters from the user...";
   call GetInputStuff;        // calls can be nested, too
}



subroutine GetInputStuff
{
   Ask ( tString, "Enter command line parameters" );
}
```

Command: **Return**

Syntax: return

Use: The Return command can be used at any time to exit the current subroutine. Normally, subroutines exit when their ending **END** (or **}**) is encountered.

Example:

```
call doSomething;


...more code...


//    ask them for their name.  If they cancel it,
//    return immediately without saying anything.
//    ----------------------------------------

subroutine doSomething;
{
   Ask ( tString, "Enter your name" );
   if ( tString = "" )
      return;
   echo "Hello there, " + tString;
}
```

Command: **ExitScript** (alias **Exit**)

Syntax: ExitScript

Use: The ExitScript command is used to halt the script processing immediately.

Example:

```
//    if an error occurred, stop execution
//    -------------------------------------

if ( error != 0 ) begin
   echo "Error " + error + " occurred..aborting!";
   ExitScript;
End;
```

Command: **Beep**

Syntax: Beep

Use: The Beep command will beep the speaker one time. This can be used to get the attention of the user.

Example:

```
//    if an error occurred, beep and put
//    out an error message
//    --------------------------------

if ( error != 0 ) begin
   for count = 1 to 3
      beep;
   echo "Error " + error + " occurred..aborting!";
   ExitScript;
End;
```

Command: **Trace**

Syntax: Trace "on" | "off"

Use: The Trace command is a script debugging tool that can be used to force each line to be displayed to the Game Window terminal screen as it is executed. By default, trace always starts out in the "off" mode. You can use multiple trace statements to toggle the state on and off.

Note: **Not yet implemented**.

Example:

```
//    Turn on tracing..
//    -----------------

Trace on

...some new code...
...some new code...

Trace off

...some code...
...some code...
```

Command: **Put**

Syntax: put value [+ value + value...]

Use: The Put command is used to send a command to the game.

Example:

```
//    Get our weapon and shield ready to go
//    -------------------------------------

put "get my " + %weapon + " from my " + %container;

put "remove my " + %shield;
```

Command: **Ask**

Syntax: ask string_variable, variable

Use: The Ask command prompts the user with a dialog box and stores what the user typed in to the specified string variable. If the user cancels the dialog box, the resulting string variable will be empty (ie: "").

Example:

```
//    If the user didn't specify a command
//    line parameter, prompt them for it.
//    -----------------------------------

string  whatToSell;

set whatToSell = %1;                // from the command line

if ( whatToSell = "" )              // did they specify it?
{
   Ask ( whatToSell, "Sell what?" );
   if ( whatToSell = "" )
   {
      exitScript;                   // they must've changed their mind
   }
}
```

Command: **Get**

Syntax: get string_variable

Use: The Get command reads the next line output by the game (that is, the information that comes across the modem). This is the command that you'll use to check on responses from your commands.

Example:

```
//    Find out how much money we've got in our
//    pockets, and deposit it all at one shot.
//    ----------------------------------------

string  input, money;

put "wealth";

while ( 1 = 1 )
{
   get input;

   if ( input == "you have no" )                // we're broke!
      break;                                    // exit the WHILE loop
   else if ( input == "you have" )
   {
      GetNumberAfter ( input, "you have", money );
      put "deposit " + money;
      break;                                    // exit the WHILE loop
   }
}
```

Command: **Wait**

Syntax: wait

Use: The Wait command pauses the script until the next command prompt is detected. This command is rarely needed.

Example:

```
//    Put out a command, and wait for a prompt.
//    ----------------------------------------

put "sit";

wait;

echo "Got a prompt!";
```

Command: **NextRoom**

Syntax: nextroom

Use: The NextRoom command pauses the script execution until a new room description is detected. This is used after "walking" commands to pause until you've completed the movement.

Example:

```
//    Walk around a bit
//    -----------------

put "n";
nextRoom;
put "e";
nextRoom;
put "go bridge";
nextRoom;
```

Command: **Move**

Syntax: move variable

Use: The Move command does the equivalent of a PUT and a NEXTROOM command. The above example (NextRoom) could be significantly shorter by using the MOVE command instead, as in:

Example:

```
//    Walk around a bit
//    -----------------

move "n";
move "e";
move “go bridge";
```

Command: **WaitFor**

Syntax: waitfor value + value + value...

Use: The WaitFor command is a shortcut command that pauses the script until the specified string is detected coming from the game. This is much less powerful than parsing the input yourself with the **get** command, but it's here mostly to maintain backward compatibility with earlier Wizard scripts.

Example:

```
//    Walk around a bit
//    -----------------

put "n";

waitfor "You move";

echo “Got the string \”You move\””;
```

Command: **Pause**

Syntax: PAUSE value

Use: The Pause command pauses the script for the specified number of seconds.

Note: You don't have to worry about "round time" pausing in your scripts. The Wizard script engine will never execute a command while the roundtime counter is greater than 0; it will enter a "pause" state automatically until the roundtime counter goes back to zero.

Example:

```
//    Stand up and equip ourselves
//    ----------------------------

put "stand";

pause 10;

... more code ...
```

Command: **GetWord**

Syntax: getWord string_value, integer_value, string_variable

Use: The GetWord command is used to pick words out of a string variable, and is very valuable for parsing data that comes back from the game. The first parameter is the string to parse, the second parameter is the number of the word you want to get, and the third parameter is the name of the string variable where you want the result placed. If you ask for a word number that doesn't exist in the string, the result will be an empty string ("").

Examples:

```
string testString;
string theWord;

set testString = "DGate rules the world!"

GetWord ( testString, 1, theWord );

// theWord now equals "DGate"


GetWord ( testString, 3, theWord );

// theWord now equals "the"

GetWord ( testString, 20, theWord );

// theWord now equals ""
```

Command: **GetNumber**

Syntax: getNumber string_value, integer_value, integer_variable

Use: The GetNumber command is identical to the GetWord command, except that it stores the numeric result into an integer variable. If you ask for a word number that doesn't exist in the string, the result will be zero.

Example:

```
string  testString;
integer theNumber;

set testString = "DGate rules the world 100% of the time!"

GetNumber ( testString, 5, theNumber );

// theNumber now equals 100


GetNumber ( testString, 20, theNumber );

// theWord now equals 0 (word doesn't exist)
```

Command: **GetWordAfter**

Syntax: getWordAfter string_value, string_value, string_variable

Use: The GetWordAfter command will find the specified string in the source string and then extract out the next word from the string, placing it in the variable you specified. The first parameter is the string to parse, the second parameter is the search string, and the third parameter is the name of the string variable where you want the result placed. If you ask for a word following a substring that isn't found, the result will be an empty string ("").

Example:

```
string testString;
string theWord;

set testString = "DGate rules the world!"

GetWordAfter ( testString, "rules", theWord );

// theWord now equals "the"


GetWordAfter ( testString, "DGate", theWord );

// theWord now equals "rules"

GetWordAfter ( testString, "abcd", theWord );

// theWord now equals ""
```

Command: **GetWordsAfter**

Syntax: getWordsAfter string_value, string_value, string_variable

Use: The GetWordsAfter command will find the specified string in the source string and then copy the remainder of the string to the variable specified. The first parameter is the string to parse, the second parameter is the search string, and the third parameter is the name of the string variable where you want the result placed. If you specify a substring that isn't found, the result will be an empty string ("").

Example:

```
string testString;
string theWords;

set testString = "DGate rules the world!"

GetWordsAfter ( testString, "rules", theWords );

// theWords now equals " the world!"


GetWordsAfter ( testString, "abcd", theWords );

// theWords now equals ""
```

Command: **GetNumberAfter**

Syntax: getNumberAfter string_value, string_value, integer_variable

Use: The GetNumberAfter command performs the same processing as the GetWordAfter routine, with one small (but important) difference. It will parse the resulting word into a numeric string, stripping out any extraneous information. It will then store that numeric value into an integer variable. This is very valuable when the word you want to get at contains dollar signs or commas.

Examples:

```
string      testString;
string      theWord;
integer     theNumber;

set testString = "Your current balance is $5,275 gold coins."

GetWordAfter ( testString, "balance is", theWord );

// theWord now equals "$5,275"


GetNumberAfter ( testString, "balance is", theNumber );

// theNumber now equals 5275
```

Command: **SubString**

Syntax: SubString string_value, integer_value, integer_value, string_variable

Use: The SubString command is used to copy part of a string into another string variable. The first parameter is the source string, the second parameter is the start position, the third parameter is the end position, and the fourth parameter is the name of the string variable where you want the result placed.

Example:

```
string bigString;
string oneCharacter;
integer x;

set bigString = "This is a very long string for testing”;

echo “The first 10 characters are as follows:”;

for x = 1 to 10
{
    SubString ( bigString, x, x, oneCharacter );
    Echo “bigString [“ + x + “] = <” + oneCharacter + “>”;
}
```

Example 2:

```
String  cardFace, cardSuit;
String  oneCard, oneSuit;

Set cardFace = “234567890JQKA234567890JQKA234567890JQKA234567890JQKA”;
Set cardSuit = “CCCCCCCCCCCCCHHHHHHHHHHHHHDDDDDDDDDDDDDSSSSSSSSSSSSS”;

//  display the 23rd card in the deck
//  --------------------------------

SubString ( cardFace, 23, 23, oneCard );
SubString ( cardSuit, 23, 23, oneSuit );

If ( oneSuit = “D” )
    Set OneSuit = “Diamonds”;
Else if ( oneSuit = “S” )
    Set OneSuit = “Spades”;
Else if ( oneSuit = “H” )
    Set OneSuit = “Hearts”;
Else
    Set OneSuit = “Clubs”;

Echo “The 23rd card in the deck is the “ + oneCard + “ of “ + oneSuit;
```

Command: **StringToInteger**

Syntax: StringToInteger string_value, integer_variable

Use: The StringToInteger command converts a string value into a numeric value and stores the result into the specified integer variable.

Examples:

```
string testString;
integer x;

set testString = "421234";
set x = 0;

echo "testString is now <" + testString + ">"
echo "x  is now <" + x + ">"

StringToInteger ( testString, x );

echo "testString is now <" + testString + ">"
echo "x  is now <" + x + ">"
```

Command: **RemovePunctuation**

Syntax: RemovePunctuation string_variable

Use: The RemovePunctuation command removes all punctuation from the specified string variable. This command comes in very handy when you need to parse words out of a string returned from the game.

Example:

```
string test;

set test = "Wizard says, \"if you want, you can follow me!\"";

// test now contains : Wizard says, "if you want, you can follow me!"
// ------------------------------------------------------------------

echo "Before removing punctuation, test is <" + test + ">"

RemovePunctuation ( test );

// test now contains : Wizard says if you want you can follow me
// ------------------------------------------------------------

echo "After removing punctuation, test is <" + test + ">"
```

Command: **Shift**

Syntax: shift

Use: The Shift command moves the command line parameters of a command script down one (that is, %2 becomes %1, etc). This is very handy for looping through all parameters that the user specified on the command line.

Example:

```
// This command line script will display all of the
// words that the user entered on the command line.
// -----------------------------------------------

while ( %1 > "" )             // while more parameters
{
   echo "Next parameter is <" + %1 + ">";
   shift;
}
echo "All done!";
```

Command: **Random**

Syntax: random (integer_variable, integer_value)

Use: The Random command generates a random number between the values of 1 and "integer_value" and stores it in the integer variable specified.

Example:

```
// Grab our weapon from our sheath, and do an act command
// 10% of the time saying we cut our finger on it.
// ------------------------------------------------------

integer pct;

put "get my " + %weapon + " from my " + %sheath;

random ( pct, 100 );

if ( pct < 11 )
{
   put "act grimaces as he fumbles with his " +
       %weapon +
       ", cutting his finger.";
}
```