WAIStation Nifty Scripting Language

Unbelievably Informative Manual, Version .001

Art Medlar
18 January 1991

The WAIStation Nifty Scripting Language is a based on that used by the White Knight telecommunications program for Macintosh computers. With only a few exceptions and additions, the commands available in WAIStation are functionally identical to those used by White Knight. The language falls into the class referred to by professional computer scientists as “krufty”. But we make no apologies. The language existed, was easy to implement, and since it was compatible, allowed us to test scripts with White Knight long before WAIStation was up to the task.

Command Syntax:

A command line is a command, followed by the zero or more required arguments, and terminated by a carriage return. The available commands are described below. If you are using an earlier version of WAIStation, you may find that not all of these commands are supported. If so, you should pick up a newer release as soon as possible.

Each line of the script is a separate command. Blank lines are ignored. A line may be commented out by surrounding it with parentheses. Each line must be commented independently; comments can not extend across multiple lines.

A label is a marker within a procedure file indicating a location which can be branched to with a GOSUB or JUMPTO command. Labels consist of a line beginning with a colon followed by any number of characters including tabs and spaces, and ending with a carriage return. Except for when a GOSUB or JUMPTO is looking for one, labels are otherwise ignored.

Command Arguments:

There are several different types of arguments to commands. If a command requires more than one argument, the arguments must be separated by commas. The following few paragraphs describe the various argument types.

A numeric constant (*num*) is any whole number, between roughly positive and negative 2 billion.

A numeric variable (*num-var*) is a letter between a and z, possibly followed by a digit between 0 and 9, and ended with a %. The case of the letter part of the variable is not significant. The following are valid numeric variables:

a1% (which is *the vary same variable as* A1%),   b%,   G2%

Numeric variables are used for storing numeric constants. If you try to store something that isn’t a numeric constant into a numeric variable, there will be a great flash of light, and WAIStation will go away. If you try to store something way too big into a numeric variable, like over 2 billion or so, not only are you probably doing something wrong, but the unexpected will probably happen as well. Although there are 264 different possible variable names, you are only allowed to have a total of 20 active, non-empty numeric variables at one time.

A numeric expression (*num-exp*) is either a numeric variable or a numeric constant.

A string constant (*str*) is a group of one or more letters, numbers, punctuation, and spaces. Spaces at the beginning or end of a string are ignored. A string can be pretty long, up to roughly the available memory of the machine you’re using. Unfortunately, the maximum length of a line in the script is 1024 characters, so it’s not easy to get one that big.

A string variable (*str-var*) is a letter, possibly followed by a digit, and ended with a $. The case of the letter is not significant. The following are valid string variables:

q1$ (which is *the vary same variable as* Q1$),   b$,   G2$

String variables are used for storing string constants. If you try to store something that isn't a string constant into a string variable, a screaming will come across the sky, and WAIStation will act in an unfriendly manner. You can store string constants of any length into a string variable. Although there are a whole lot of possible names for string variables, you are only allowed to have 20 non-empty, active string variables at one time. This is in addition to the number of possible numeric variables; the total number of both kinds allowed at one time is thus 40.

A boolean (*boolean*) constant is one of TRUE or FALSE. YES is synonymous with TRUE and NO is synonymous with FALSE,. Sometimes that makes things a tad more readable.

There is a special flag, straightforwardly called the yes/no flag, which is set and examined by a couple of commands. Its default state is undefined, so you should probably use a command to set it before you use one to examine it. Once it is set, however, it remains set until either it is reset or the script finishes running.

Error Handling:

Currently, error handlingis awfully primitive. In most cases, errors are signalled with a little OK dialog box, and then WAIStation bites the wax tadpole. Someday, there might be something better, but until then you are advised simply to not make mistakes.

COMMANDS:

Input and output commands:

TYPE *str-exp*

Str-exp is sent to the modem. If *str-exp* is a string or numeric variable, that is, one or two alphanumeric characters followed by a $ or %, and the variable has already had something stored into it, then its contents will be sent instead. If the variable is empty, then the name of the variable will be typed. This is a not entirely elegant way of handling the situation where the user wants a literal string like, “a%”, to be typed.

PROMPT *str-exp*

The procedure waits until a string matching *str-exp* is received by the modem. The string must be matched exactly, upper and lower case are significant.

QUERY *str-var, str-exp*

A dialog box containing *str-exp* as its prompting text, a cancel and an OK button, and an area for text input appears. Text entered is stored into *str-var* when the OK button is clicked. This command is particularly useful with systems requiring passwords. The command:

QUERY a$, What is your password?

will store the users typed password into the variable a$, which can then subsequently be TYPE’d to the remote system. Since this command requires user input, it should not be used in scripts for sources that will be automatically updated, since no one is likely to be around when the automatic updation takes place.

.i.ALERT1 ;str-exp/proc-cmd
.i.ALERT2 *str-exp/proc-cmd*
ALERT3 *str-exp/proc-cmd*

Alerts are a nice way of waiting for more than one possible thing to be returned . In fact, you can wait for three things in addition to a PROMPT. While the PROMPT command is waiting for an expected string, sometimes something slightly less likely comes back. If the less likely thing matches one of the *str-exp*’s, the *proc-cmd* following the / is executed. Alerts, if any, must come before the PROMPT command. If *proc-cmd* is GOSUB, and the *str-exp* is matched, the subroutine will return to the line following the PROMPT command. When an ALERT or PROMPT is matched, all pending alerts are cleared.

PANICAFTER *num-exp*

If *num-exp* seconds pass while waiting for a prompt, a panic condition occurs, and the *proc-cmd* argument to the currently active ONPANIC command is executed. If there is no currently active ONPANIC command, nothing happens. An ONPANIC command is active if it has been called since the last successful matching of a prompt or alert. When a prompt or alert is matched, all ONPANIC and PANICAFTER commands are cleared.

ONPANIC *proc-cmd*

If a panic condition occurs during execution of a PROMPT command or while an ALERT is active, *proc-cmd* is immediately executed. *Proc-cmd* is cleared whenever a prompt or alert is successfully matched.

JUMPTO *label*

An immediate branch to *label* happens. If *label* does not exist, an error is signalled.

GOSUB *label*

An immediate branch to *label* is performed. Execution continues until a RETURN statement is encountered, which then causes a branch back to the statement immediately following the GOSUB. Subroutines can be nested 20 deep, and can be called recursively. If *label* does not exist, an error is signalled.

RETURN

This command must only occur during a branch to a subroutine within the script. Appearing anywhere else signals an error. Execution branches back to the line immediately following the most recently called GOSUB.

IF **YES** *proc_cmd*

If the yes-no flag is set to YES, then *proc-cmd* is executed, otherwise it is not.

IF **NO** *proc-cmd*

If the yes-no flag is set to NO, then *proc-cmd* is executed, otherwise it is not.

Arithmetic Commands:

AND *num-var, num-exp*

Does a binary AND to the value of *num-exp* and the value contained in *num-var*, and stores the result into *num-var*. *Num-exp* is not affected.

OR *num-var, num-exp*

Does a binary OR to the value of *num-exp* and the value contained in *num-var*, and stores the result into *num-var*. *Num-exp* is not affected.

ADD *num-var, num-exp*

Adds the value of *num-exp* to the value contained in *num-var* and stores the result in *num-var*. *Num-exp* is not affected.

SUBTRACT *num-var, num-exp*

Subtracts the value of *num-exp* from the value contained in *num-var* and stores the result in *num-var*. *Num-exp* is not affected.

MULTIPLY *num-var, num-exp*

Multiplies the value of *num-exp* with the value contained in *num-var* and stores the result in *num-var*. *Num-exp* is not affected.

DIVIDE *num-var, num-exp*

Divides the value of *num-var* by the value contained in *num-exp* and stores the result in *num-var*. It's anybody's guess what happens if the value of *num-exp* is Zero. *Num-exp* is not affected.

BYTEVAL *str-var, num-exp, num-var*

The ASCII value of the *num-expth* byte of *str-var* is stored in *num-var*. No checking is done to determine if, in fact, there is a *num-expth* byte of *str-var*. No error is signaled if there isn't.

BYTEADD *num-exp, str-var*

Bungs on a character with ASCII value *num-exp* to the end of the string contained in *str-var*. Heaven knows why you would ever need to do this.

TEST *num-var, num-test-op, num-exp*

An arithmetical test is performed between the contents of *num-var* and the value of *num-exp*. The result of this test is used to set the yes/no flag. *Num-test-op* can be one of <, >, >=, <=, <>, &, |, all of which have the obvious meaning. If the test is true the yes/no flag is set to YES, otherwise NO. This command does not require commas between the arguments, they can be replaced by spaces. Spaces don't make things ambiguous and do make it a lot easier to read. *Num-var* and *Num-exp* are not affected by this operation.

String Manipulation Commands:

CONCAT *str-var, str-exp*

The string *str-exp* is bunged onto the end of the string contained in *str-var* and the result is stored back into *str-var*. *Str-exp* is not affected by this operation.

CONTAINS *str-var, str-exp*

If the string contained in *str-var* contains *str-exp*, the value of the yes/no flag is set to YES. Otherwise it is set to NO. *Str-var* and *str-exp* are left unchanged. This operation is case-sensitive.

CONVUP *str-var*

The string contained in *str-var* is converted to upper-case.

COPYINTO *str-var, str-exp*

The contents of *str-exp* are copied into *str-var.* The old contents of *str-var*, if any, are lost, the old contents of *str-exp* are left unchanged.

EMPTY *str-var*

If *str-var* contains no characters, the yes/no flag is set to YES, otherwise it is set to NO.

LENGTH *num-var, str-exp*

The number of characters in *str-exp* is stored into *num-var.*

LET EQUAL *num-var, num-exp*
The value of *num-exp* is stored into *num-var.*

NUMTOSTRING *num-var, str-var*
The numeric value in *num-var* is converted into a string and stored into *str-var.*

STRINGTONUM *str-var, num-var*
The string of numerals in *str-var* is converted into a number and stored into *num-var.* If there is something in *str-var* which is not a numeral, and therefore impossible to convert into a number, a big error happens.

ERASE *str-var*
ERASE all
The former empties the contents of *str-var*, the latter empties the contents of all currently defined string variables.

Miscellaneous Commands:

ELAPSED *num-var*
The number of seconds which have passed since the last SAVETIME command is stored into *num-var.*

SAVETIME
This command saves the current time internally. It is used in conjunction with the ELAPSED command to determine how may seconds have passed.

COMM *str-exp*
Str-exp is of the form BAUD-PARITY-DATABITS-STOPBITS-DUPLEX. For example: COMM 9600-N-8-1-FULL. This changes the communications parameters of the modem.

LONG BREAK, **SHORT BREAK**
These send either a long (4 second) or short (233 millisecond) break signal to the modem. Check your modem manual to determine what happens when it gets there.

COMMENT
The remainder of the line is ignored by the script interpreter. Alternatively, parentheses placed around a line hide it from the interpreter. Comments do not extend for multiple lines. Each line must be commented out separately.

PAUSE *num-exp*
Execution of the script pauses for *num-exp* seconds.

CRAWL **on/off**
The interpreter pauses for 2 seconds between execution of command lines between CRAWL ON and CRAWL OFF. This is sometimes more convenient than putting a PAUSE between each line.

LOUD
Procedure execution is monitored in a special window. The special window has a way of popping up right on top of everything, and just generally getting in the way, so you'll have to play around a bit before this command seems more useful than annoying.

QUIET
Procedure execution stops being monitored in a special window. The special window does not disappear. This is too bad, but it cannot be helped.

END
Execution of the script is terminated.

Special Commands:

CHUNKING *boolean*
This is true if you are attempting to connect to one of those broken types of servers which cannot handle more than a small number of bytes at a time. Frequently, there will be some IBM product between you and your search engine, and you will need to set this to TRUE. The default is FALSE.

CHUNQUETTESIZE *num*
In case CHUNKING is TRUE, this must be set to the size of chunks into which the outgoing data will be divided.

ECHOING *boolean*
This should be TRUE if the WAIS server will be echoing packets sent by the WAIStation. For reasons of performance and efficiency, you should make every effort to get an echoing server to stop echoing. Since that can't always be done (often there's some IBM product between you and your search engine, for example) this command corrects for such server design flaws.

HEXING *hex-type*
Currently there are three types of encoding used in the transport layer underneath the z39.50 protocol. The options are IBM, OLD, and NEW. This is only a suggestion to WAIStation as to which one to use. It will try to adapt to whatever type the server is using. In most cases, the value of *hex-type* should be IBM.

SCRIPT_TIMEOUT *num*
This is the maximum number of seconds waited for a prompt or alert to be returned before terminating script execution. *Num* should be greater than any pending PANICAFTER time, otherwise the panic condition will never occur.

DEBUGGING *boolean*
Set this to TRUE to watch what's happening with the modem and to log all incoming and outgoing z39.50 packets into the file named with the LOGFILE command.

LOGFILE *pathname*

Used in conjunction with the DEBUGGING command. *Pathname* is of the form volume:folder:folder....:filename. Filename need not already exist. If it does, the former contents are destroyed. It is very important that all folders exist, though. If they don't, WAIStation will swiftly and suddenly vanish away, and never be met with again.

DOWQUEST

If you are attempting to connect to DowQuest, this command should appear somewhere in the script. It sets an internal variable which compensates for a number of, shall we say politely, less than optimal implementation characteristics of the DowJones system.

INDEX: