Amazon Reader

Operation

Amazon Reader is a program for reading a spreadsheet file formatted to hold Amazon purchase information with each row indicating an item that was purchased and the columns representing pertanent information about the the order. It allows the user to then read in from the system clipboard web pages that are clipped from the Amazon orders site and then extracts the desired information from them, and places that data into the appropriate columns of spreadsheet image, prior to saving the image back to the spreadsheet file. It also allows reading the text data from a PDF file containing the credit card charges from Amazon for the purchases and again marks the appropriate columns with the charge information to verify which items have been completed and which are still pending, along with the credit card file id that the charge information came from. This allows a semi-automated process for keeping track of the Amazon orders and their corresponding debits and credits that are charged to the credit card to verify there are no unknown charges received.

The program can be run in 3 different fashions, but for normal use of balancing the Amazon charges the GUI interface is used. This is performed by simply running the jar file with no arguments:

```
java -jar AmazonReader-1.2.jar
```

The command should be run from the parent directory of where all the spreadsheet files for Amazon are kept. This is because it also creates a Properties file that will keep track of the directories and settings selected, so that when you start the program it will automatically use those settings so you don't have to set them up again. The details of the Properties file will be explained in a later section.

The program also contains some additional test commands that run pieces of the code for test verification. These are not accessible from the GUI, but are provided by using Command-line Options. When running from the command line, the program will only perform the actions of the Options that are supplied on the command line. An example of this would be:

```
java -jar AmazonReader-1.2.jar -s testfile.ods -l 1 true -c clip1.txt -u -save
```

Note that multiple options (each option command starts with a '-' char) can be placed in a single command line. This command, for instance, would select the spreadsheet file 'testfile.ods' and load the 1st tab into memory while performing a verification that the header information contained in the file is valid. Then it would read in and parse the information from the clipboard file 'clip1.txt' (rather than reading directly from the clipboard) and update the the spreadsheet image with any new entries it gathered from it. It would then save the updated image back to the spreadsheet file.

For more extensive testing, it is rather cumbersome to write all the option commands you want to perform on a single command line. But because the command line operation executes the command options as given and then exits, there is no way to execute sequential commands where the next command depends on the previous command, unless each commad only depends on the state of the spreadsheet file and you make sure to save changes back to the file at the end of each command to make sure the next one gets those changes. This can be done, but is a slow process since it takes a several seconds for each saving and reloading of the spreadsheet file. This can be better accompplished by using the Program operation mode. In this mode, you create a script file of what you want to execute, then pass that as the only argument to the program. It will then execute all the program statements sequentially before exiting. There are also program flow statements that allow you to perform loops and conditionals, as well as variable parameters for manipulating any data captured and testing it for validation. The format for this operation is:

```
java -jar AmazonReader-1.2.jar -f myscript.scr
```

If you want to simply run the compiler on the script file to check for errors without actually executing any script statements, use the following format:

```
java -jar AmazonReader-1.2.jar -c myscript.scr
```

Properties File

The Properties file is a file that is used to keep track of settings that are made with the command options so that it can remember these settings from previous calls. It is created (and updated) when you run AmazonReader.jar and will be placed in the directory you execute the AmazonReader.jar file from (NOT where the jar file exists, but where you run the command from). This way, each location you execute from can have its own set of parameters that it remembers. The file is a hidden file in a hidden directory called: .amazonreader/site.properties. It contains a list of the settings to be maintained each time the program is run with each having 2 parts: the identifier tag and the value. An example file is as follows:

```
#---No Comment---
#Sun Mar 30 10:01:38 EDT 2025
MaxLenDescription=100
DebugFileOut=debug.log
MsgEnable=0x2F
PdfPath=/home/dan/Records/Finance/Credit_card_statements/2025/Chase_VISA_3996
SpreadsheetPath=/home/dan/Records/Finance/Amazon/Testing
SpreadsheetFile=NewTest.ods
SpreadsheetTab=0
TestPath=/home/dan/Records/Finance/Amazon/Testing
TestFileOut=test.log
```

Note that it contains a comment line that shows the last date and time the file was updated. Every time one of the settings is changed by the running program. The definitions are:

`MaxLenDescription`

Defines the maximum character length to allow for the Description field when extracting content from the clipboard to the spreadsheet.
(not set by any command, must be set manually)

`DebugFileOut`

Defines the file name to output debug messages to in GUI mode. The file path is set to the value of `SpreadsheetPath`. If none, output will go to standard out.
(not set by any command, must be set manually)

`MsgEnable`

Selects the debug messages that are enabled
(set by the `-debug` option and when the message checkboxes are selected on the GUI panel)

`PdfPath`

Defines the path to read the PDF file from. If not defined, it will use the current directory.
(set by the `-pfile` option and when the *Balance* button is pressed from the GUI to load the PDF file)

`SpreadsheetPath`

Defines the path to read the Spreadsheet file from. If not defined, it will use the current directory.
(set by the `-sfile` option if an absolute path is specified and when *Select* button is pressed from the GUI)

`SpreadsheetFile`

Defines the name of the Spreadsheet file to read
(set by the `-sfile` option and when *Select* button is pressed from the GUI)

`SpreadsheetTab`

Defines the starting tab number of the Spreadsheet file to load (0 for 1st tab).
(set by the `-tab` option directly and and by the `-update` or `-pfile` options (or the GUI when *Update* or *Balance* buttons are pressed) when updating the spreadsheet from the Clipboard or PDF file.

`TestPath`

Defines the path of used for locating the script file to run and the base path for the `TestFileOut` and the File Commands. This allows you to run from a different directory than where the script resides.
(set by the `-tfile` option)

`TestFileOut`

Defines the file name to output debug messages to when running from a program script. If blank or omitted, output is directed to stdout.
(set by the `-ofile` option)

Data Format Descriptions

The following indicates the format for the different categories of elements used in creating a program. The brown color represents hardcoded character values and green indicates one of the other defined data types. Braces { } can hold multiple values, each seperated by a pipe (‘|’) character. Optional elements are indicated by red type. If multiple occurrances of elements are permissable, a subscript will follow the bracket containing the number of the number of repititions allowed. Note that the braces, brackets, pipes and subscripts are not part of the format, they are just for conveying info about te format. Also, when an optional DblQuote or ParenLeft is used, the corresponding ending DblQuote or ParenRight must also be used.

Basic definitions

`Printable`	`ASCII char 0x21 , 0x23 – 0x7E`	`(exclude space and double quote)`
`WS`	`{ ASCII char 0x20 }`$_{1-N}$	`(whitespace)`
`Underscore`	`{ _ }`	
`Dash`	`{ - }`	
`Comma`	`{ , }`	
`DblQuote`	`{ “ }`	
`LParen`	`{ ( }`	
`RParen`	`{ ) }`	
`LBracket`	`{ [ }`	
`RBracket`	`{ ] }`	
`MathOp`	`{ + | - | * | / | % }`	
`BitOp`	`{ AND | OR | XOR | NOT | ROR | ROL }`	
`CompSign`	`{ == | != | > | < | >= | <= }`	
`DecDigit`	`{ 0-9 }`	
`HexDigit`	`{ 0-9 | A-F | a-f }`	
`LowerAlpha`	`{ a-z }`	
`UpperAlpha`	`{ A-Z }`	
`Alpha`	`{UpperAlpha | LowerAlpha}`	
`NumericHex`	`0x  {HexDigit}`$_{1-8}$	
`NumericDec`	`- {DecDigit}`$_{1-10}$	
`NumericLong`	`- {DecDigit}`$_{1-19}$	
`UnquotedStr`	`{Printable}`$_{1-N}$	
`QuotedStr`	`“ {Printable | WS}`$_{0-N}$ `“`	

Simple Data type definitions

`Boolean`	`{ TRUE | FALSE | true | false | 1 | 0 }`
`Integer`	`{NumericLong | NumericDec | NumericHex | ParamRef}`
`Unsigned`	`{NumericDec | NumericHex | ParamRef}`
`String`	`{UnquotedStr | QuotedStr | ParamRef}`

Multi-element Data type definitions

(whitespace is ignored between elements)

`IntArray`	`Integer { , Integer }`$_{0-N}$
`StrArray`	`String  { , String }`$_{0-N}$

Left-side entity definitions

```
Cmd              {UpperAlpha}_1-N                                 (from a defined list)
Option           - LowerAlpha {LowerAlpha | DecDigit}_1-N         (from a defined list)
ParamName_STR    {Alpha} {Alpha | DecDigit | _ }_0-19
ParamName_INT    I_ {Alpha | DecDigit | _ }_1-18
ParamName_UNS    U_ {Alpha | DecDigit | _ }_1-18
ParamName_BOOL   B_ {Alpha | DecDigit | _ }_1-18
ParamName_SARR   L_ {Alpha | DecDigit | _ }_1-18
ParamName_IARR   A_ {Alpha | DecDigit | _ }_1-18
```

Right-side entity definitions

```
ParamValue       {Integer | Unsigned | Boolean | String | IntArray | StrArray}
ParamRef         $ ParamName
                 $ ParamName_STR [  NumericDec [ - NumericDec ] ]                (returns partial String)
                 $ ParamName_SARR [  NumericDec ]                                (returns String entry)
                 $ ParamName_IARR [  NumericDec ]                                (returns Integer entry)
                 $ { ParamName_STR | ParamName_SARR | ParamName_IARR }.LENGTH    (returns Integer length)
                 $ { ParamName_STR | ParamName_SARR | ParamName_IARR }.ISEMPTY   (returns Boolean)
                 $ ParamName_STR . { UPPER | LOWER }                             (returns converted String)
                 $ ParamName_SARR . { FIRST | LAST }                             (returns String entry)
                 $ ParamName_IARR . { FIRST | LAST }                             (returns Integer entry)
CalcSegment      ( Integer {MathOp Integer}_1-8  )                               (for Integers)
                 ( Unsigned { {MathOp | BitOp} Unsigned}_1-8  )                              (for Unsigned)
Calculation      { ( {CalcSegment} ) }_1-8
Comparison       Calculation  CompSign  Calculation
```

Command line formats

(whitespace is ignored between elements in a command)

```
General cmd      {Cmd}  {ParamValue}_0-N
FOR cmd          FOR     ParamName = Integer ; CmpSign ParamValue ; Integer
IF cmd           IF      ParamName  CompSign  ParamValue
WHILE cmd        WHILE   ParamName  CompSign  ParamValue
OptCommand       RUN     { Option {ParamValue}_0-N }_0-N
Assignment       SET     ParamName = Calculation
```

Command Line Mode

Command-line Options are defined for performing some of the Amazon Logger actions. Some have associated arguments and others do not, but all are denoted by beginning with a '-' character. This is because they are also available as command line arguments as well as being used in a program script. They can be concatenated in a command line. That is, you can place more than one command line option in one line of the script. In the *Option* command list that follows, the list of arguments is indicated and each argument will begin with a character and an underscore (_). The leading character of the argument name will indicate the data type of the argument allowed.

Note that all files are referenced from the current test path that is specified in the Properties file. The definitions for the data types above are:

U – Unsigned integer 32-bit

I – Integer 64-bit (signed)

B – Boolean (true or false, or 1 for true and 0 for false)

S – String

L – List of Strings (array)

A – Array of Integers

`-debug` ***U****_flags*

Sets the debug messages that are enabled. Note that the ERROR and WARN messages are always enabled, regardless of the debug flag value. Also, ERROR conditions will terminate the program without completing any further operations. The default value for the message selection will be pulled from the `MsgEnable` entry in the Properties file entry, so it will remember the last value that was set, as long as you run from the same directory.

The following is a list of the errors:

Name	Hex value	Dec value	GUI color	GUI font	Description
NORMAL	0x0001	1	Black	Normal	Changes made to spreadsheet
PARSER	0x0002	2	Blue	Italic	Parser status conditions
SSHEET	0x0004	4	Green	Italic	Spreadsheet status conditions
INFO	0x0008	8	Dk Violet	Normal	Web clip and PDF file processing info
PROPS	0x0010	16	Gold	Italic	Properties file changes
PROGRAM	0x0020	32	Dk Violet	Normal	Program interface messages
DEBUG	0x0800	2048	Brown	Normal	Low-level debugging details
WARN	0x4000	16384	Orange	Bold	Non-fatal warnings
ERROR	0x8000	32768	Red	Bold	Fatal errors

`Examples:`

To enable only the PROGRAM, DEBUG, and PROPS flags, this would have a combined value of 0x20 + 0x800 + 0x10 = 0x830 or decimal 2096, so you could do either of the following:

```
-debug 0x830
-debug 2096
```

`-sfile S_file`

Sets the Spreadsheet filename to load (must be an .ods file). This value gets saved to the `SpreadsheetFile` entry in the Properties file entry, so it will use this last setting when you start AmazonReader if you omit sending this command. The filename can either be absolute or relative. If you specify the directory by giving the absolute pathname (it must start with either the '/' or the '~' character, where '/' specifies the root path and '~' specifies the home directory), the path portion of the name will be saved in `SpreadsheetPath` and the remainder in `SpreadsheetFile` in the Properties file. If no entry is found for `SpreadsheetPath` in the Properties file or no Properties file currently exists, and if the absolute path was not specified in the command, the current user directory (where the jar command is being invoked from) will be used as the path to base the file selection on.

`-load U_numTabs B_chkHeader`

Loads the selected Spreadsheet file tabs into memory (if chkHeader true, verify the header format is valid and classify the columns for use). All operation work on this *image*.

`-tab U_tab`

Sets the tab of the current Spreadsheet to use in operations

`-cfile S_file`

Reads the Clipboard file selection (uses a file instead of clipboard in reading Amazon orders). This uses the `TestPath` path from the Properties file to base the path on if the file is specified as a relative path. If this entry is not found in the Properties file or no Properties file currently exists the current user directory (where the jar command is being invoked from) will be used as the path to base the file selection on.

`-clip B_strip`

Reads the system clipboard and saves the lines in `$RESPONSE` array. If the optional `B_strip` argument is given, a value of `TRUE` will cause it to strip all leading and trailing whitespace from the lines and will eliminate any empty lines. By default, the clipboard contents will not be modified.

`-update`

Updates the spreadsheet from the clipboards

`-pfile S_file`

Selects and loads the PDF file to read and saves the non-empty lines in `$RESPONSE` array. The directory portion of the value gets saved to the `PdfPath` entry in the Properties file entry, so it will use this last setting when you start AmazonReader if you omit sending this command. If no absolute path is specified, it will be relative to the `PdfPath` in the Properties file. If this entry is not found in the Properties file or no Properties file currently exists the current user directory (where the jar command is being invoked from) will be used as the path to base the file selection on.

`-prun`

Processes the PDF file loaded by the `-p` option to read and mark off the items processed in the spreadsheet *image*

`-ofile S_file`

Sets the file to output the debug messages to (`S_file` is optional: if omitted, msgs go to stdout)

`-tpath S_path`

Sets the location of where the script file is to be executed from (in case you are running from a different directory than the script file) and where the debug output file and the File Commands are referenced from. (`S_path` is optional: if omitted, the current user directory you are executing from will be used)

`-save`

Saves the current image info to the spreadsheet file and reloads the file back into memory

`-date` ***L**_dateFields*

Converts the list of fields into a date (assumes the date is in the future if a relative date given)

This command will return the response in the `$RESPONSE` array

`-datep` ***L**_dateFields*

Converts the list of fields into a date (assumes the date was in the past if a relative date given)

This command will return the response in the `$RESPONSE` array

`-default` ***U**_numTabs* ***B**_chkHeader*

Sets the Spreadsheet path, file and tab selections from the Properties File settings (which will be set to the last values previously used) and then loads the spreadsheet using the same arguments as the `-l` option. This keeps you from having to specify the `-s` and `-t` options every time you want to load a spreadsheet file.

`-maxcol`

Returns the number of columns defined in the current spreadsheet tab selection in the `$RESPONSE` array

`-maxrow`

Returns the number of columns defined in the current spreadsheet tab selection in the `$RESPONSE` array

`-setsize` ***U**_cols* ***U**_rows*

Resizes the spreadsheet image to the specified number of columns and rows

`-find` ***S**_orderNum*

Finds the first row containing the specified Amazon Order Number and returns it in the `$RESPONSE` array

`-class` ***U**_cols* ***U**_rows*

Gets the Class type of the cell in the current tab selection of the current spreadsheet image at the specified column and row location and returns it in the `$RESPONSE` array

`-color` ***U**_cols* ***U**_rows* ***U**_Mocolor*

Sets the background color of the specified spreadsheet image cell location to the specified color of the month (*U_MOcolor* must be a value from 1 to 12 for the month, 0 for white, or -1 for black)

`-RGB` ***U**_cols* ***U**_rows* ***U**_RGBcolor*

Sets the background color of the specified spreadsheet image cell location to the specified RGB color (6 hex digits representing 2 digits Red, 2 digits Green, 2 digits Blue)

`-HSB` ***U**_cols* ***U**_rows* ***U**_HSBcolor*

Sets the background color of the specified spreadsheet image cell location to the specified color of the month (6 hex digits representing 2 digits Hue, 2 digits Saturation, 2 digits Brightness)

`-cellclr` ***U**_cols* ***U**_rows*

Clears the text contained at the specified cell location of the current spreadsheet image

`-cellget` ***U**_cols* ***U**_rows*

Gets the text data contained in the specified cell location of the current spreadsheet image and returns it in the `$RESPONSE` parameter

`-cellput` ***U**_cols* ***U**_rows* ***S**_text*

Sets the text at the specified cell location of the current spreadsheet image to the fields specified

`-rowget` ***U**_col* ***U**_row* ***U**_count*

Gets the text data contained in the specified number of cells in the selected row starting from the specified column and returns it in the `$RESPONSE` array. The row will be ***U**_row* and the columns will be from ***U**_col* to ***U**_col* + ***U**_count*.

`-colget` ***U**_col* ***U**_row* ***U**_count*

Gets the text data contained in the specified number of cells in the selected column starting from the specified row and returns it in the `$RESPONSE` array. The column will be ***U**_col* and the rows will be from ***U**_row* to ***U**_row* + ***U**_count*.

`-rowput` ***U**_col* ***U**_row* ***L**_array*

Sets the text data contained in the row of cells to the data contained in the array. The first entry in the array will be placed at ***U**_row* , ***U**_col* and the last entry at ***U**_row* , ***U**_col* + sizeof(***L**_array*) - 1.

`-colput` ***U**_col* ***U**_row* ***L**_array*

Sets the text data contained in the column of cells to the data contained in the array. The first entry in the array will be placed at ***U**_row* , ***U**_col* and the last entry at ***U**_row* + sizeof(***L**_array*) - 1, ***U**_col*.

`-rowcolor` ***U**_col* ***U**_row* ***A**_array*

Sets the background color of the row of cells to the data contained in the array. The first entry in the array will be placed at ***U**_row* , ***U**_col* and the last entry at ***U**_row* , ***U**_col* + sizeof(***A**_array*) - 1.

`-colcolor` ***U**_col* ***U**_row* ***A**_array*

Sets the background color of the column of cells to the data contained in the array. The first entry in the array will be placed at ***U**_row* , ***U**_col* and the last entry at ***U**_row* + sizeof(***A**_array*) - 1, ***U**_col*.

Program Mode

Statements

A program conssts of a series of statements, some of which are for performing specific actions, some are for control of the program flow, and others are for documenting the operation being performed.

There are 4 types of statements that are allowed:

- Comments
- Command-line Options (see previous section for details)
- Program Commands
- Parameter Assignments

Comments

Comments are any statements that begin with the '#' character (leading whitespace is allowed). You can also add a comment at the end of a command line by placing "##" characters at the beginning of the comment secion. Comments are ignored by the compiler.

```
Examples:
       # This section of code is used to remove entries from an array
       #  without overrunning the array size, which would result in an error.
       # This isn't the best way to do this, but is for demonstration purposes.
       FOR I_Index = 1 ; <= 10 ; 1          ## perform 10 iterations
           IF NOT $A_NumArray.ISEMPTY       ## check first that the array is not empty
               POP $A_NumArray              ## if not, remove the 1st entry in the array
           ENDIF
       NEXT
       ENDFOR                               ## ends the loop
```

Program Commands

Note that the program commands are always uppercase only. Optional arguments are indicated in red. If there are different types of arguments that are allowed, they will be enclosed in braces { } and separated by the pipe character, |. Multiple repititions of arguments are represented by a subscript of the number of occurrances allowed.

DEFINE *ParamName , ParamName*$_{0-N}$

Defines one or more parameters to be used in the script. Parameters are comma separated if more than one is defined and should be enclosed in braces. All parameters must be defined before they can be used. Note that the parameter names define their type, so refer to the Parameters section for more details. *Integer* and *Unsigned* types are initialized to 0, *Boolean* types to FALSE, *Strings* to and empty String, and *Array* types to an empty array.

```
Examples:
       DEFINE String1
       DEFINE { String2, I_Counter, A_NumericArray }
```

SET *ParamName = Calculation*

Refer to the Parameter Assignment statement for details (SET keyword is optional). Note that if you don't use quotes to enclose a String value, you must confine it to a single word (no spaces). Also, the Variable names are expressed on the left side of the equals sign do not use the '`$`' character, but must be used on the right side. It is meant to indicate it is a *reference* to a variable.

```
Examples:
        SET String1 = SingleWord
        SET String2 = “hello world!”
        B_Status = TRUE
        A_NumericArray = { 143, -22, 777, 0xFFFF, 999, 222, 87654, -12 }
        I_ROW = ($STRVAL % 25) - (40 + (6 * ($I_COL + 5) / 10)) * -3 - 8
```

RUN *Option*

This will run the specified Command Option (or Options, as it can have sequential options in a single command). Refer to the *Command Line Mode* section for more information of the Command Options.

```
Examples:
        RUN -s TestSheet.ods
        RUN -colput $I_COL $I_ROW { 100, 200, 300, 400, 500 }
```

IF { *ParamName*$_{INT}$ | *ParamName*$_{UNS}$ } *CompSign Calculation*
IF { *ParamName*$_{STR}$ } *CompSign* { *ParamName*$_{STR}$ | *String* }
IF { *NOT* | *!* } { *ParamName*$_{BOOL}$ | *Comparison* | *Boolean* }

This is the basic conditional statement. It will verify whether the expression is true or not. If so, the program proceeds to the next command line. If it is not true, it will proceed to the next `ELSE`, `ELSEIF` or `ENDIF` statement it finds. The *`CompSign`* value is one of { `==`, `!=`, `>=`, `<=`, `>`, `<` }. The *`Calculation`* is an algebraic statement that can be composed of parameters, numbers, operations and parantheses. The operations allowed are: { `+`, `-`, `*`, `/`, `%` }. For String comparisons, it will only take a String value (quoted if more than a single word) or another String parameter. Boolean comparisons take either a Boolean parameter, Boolean value, or a comparison expression that will equate to a `TRUE` or `FALSE` condition. Using a discrete `TRUE` or `FALSE` value will allow forcing it to always or never branch, which can allow temporarily branch forcing during testing.

If the condition was met and the code following the IF statement is executed, it will continue to execute sequential commands until it gets to an `ELSE` or `ELSEIF` statement, which will cause it to jump to the next `ENDIF` statement. IF statements can also be nested inside each other.

```
Examples:
        IF $A_NumArray[20] >= 730     ## checks the value of the 21st entry of the numeric array
        IF $A_NumArray.SIZE < 10      ## checks if the numeric array has less than 10 entries
        IF $String1 == $String2       ## checks if the 2 String variables are equal
        IF $B_Flag                    ## checks if the Boolean variable is TRUE
        IF $L_STRArray.ISEMPTY        ## checks if the String array is empty (0 elements)
```

ELSEIF *ParamName CompSign Calculation*

This is the same as the `IF` statement in operation, but as an alternate condition if the previous `IF` or `ELSEIF` condition was not met. If the previous condition was met, this will cause the program to jump to the next `ENDIF` statement. Otherwise, it will test the new condition to see if this one is valid or not. If valid, program flow will continue with the subsequent line. Otherwise, it will search for the next `ELSE` or `ELSEIF` statement.

```
Examples:
        IF $I_Count >= 730
            I_Count -= 100
        ELSEIF $I_Count >= 220
            I_Count += 100
        ELSE
            I_Count += 10
        ENDIF
```

ELSE

This takes no arguments and performs no action. It is simply a marker for where to redirect program flow if the condition wasn't met, and an indication to jump to the `ENDIF` if the previous condition was met. This also completes the conditionals, so there can be no further `ELSE` or `ELSEIF` following this command for the current `IF` level.

ENDIF

This takes no arguments and performs no action. It is simply a marker for where to jump to when the previous conditions have concluded.

FOR *ParamName = InitValue ; CompSign EndValue ; StepValue*

This allows a series of commands to be repeated for a specified number of times. It has 3 argument sections, each seperated by a semicolon. The format of the arguments is given by the `ForParams` entry in the *Data Format Descriptions* section. The first is the loop parameter name followed by and equals sign and the *`InitValue`* to initialize it to for its first pass through the loop. This can be either a hard-coded value or a parameter reference. The second section specifies the *`EndValue`* for the loop parameter – in other words, when this condition is met the loop will be exited. The *`CompSign`* value is one of { ==, !=, >=, <=, >, < }. The third and final section specifies *`StepValue`*, the increment size of the loop parameter (how much to add to it after each loop) and is optional. This value can be a positive or negative value and can also be a parameter reference. If omitted, the step size is an increase of 1. The loop name follows the same rules as for the String parameter, but must be different than any parameter name. Loops can also be nested, but the loop parameter name must be different for each loop. The loop parameter name can be reused once outside its current loop. Note that the loop will not run at all if the *`InitValue`* exceeds the conditions of the *`EndValue`*.

```
Examples:
       FOR I_Index = 1 ; <= 10 ; 1   ## will perform 10 iterations
           IF $I_Count >= $I_Index
               PRINT $I_Count        ## if I_Count >= I_Index, will print value of I_Count
               BREAK                 ##  then will go to ENDFOR statement to exit loop
           ELSEIF $I_Count < 10
               CONTINUE              ## if I_Count < 10, will go to NEXT statement
           ENDIF
       NEXT                          ## adds 1 to I_Index and return to FOR statement
       ENDFOR                        ## ends the loop & proceeds to subsequent statement
```

BREAK

This is used within the most recent `FOR` loop and will cause the program to jump to the next `ENDFOR` statement.

CONTINUE

This is used within the most recent `FOR` loop and will cause the program to jump to the following `NEXT` command, which will increase the loop parameter by the *`StepValue`* and then test to see if the *`EndValue`* condition has been met. If so, it will jump to the next `ENDFOR` statement, otherwise it will jump back to the beginning of the loop.

NEXT

This is used within the most recent `FOR` loop and will cause the program to increase the loop parameter by the *`StepValue`* and then test to see if the *`EndValue`* condition has been met. If so, it will jump to the next `ENDFOR` statement, otherwise it will jump back to the beginning of the loop.

ENDFOR

This takes no arguments and performs no action. It is simply a marker for where to jump to when the loop has completed.

Array-only commands

These commands are only available to *StrArray* and *IntArray* parameters.

`INSERT` *ParamName*$_{SARR}$ *String*

`INSERT` *ParamName*$_{IARR}$ *Integer*

This command will insert the specified value at the beginning of the array.

`APPEND` *ParamName*$_{SARR}$ *String*

`APPEND` *ParamName*$_{IARR}$ *Integer*

This command will append the specified value at the end of the array.

`MODIFY` *ParamName*$_{SARR}$ *DecValue String*

`MODIFY` *ParamName*$_{IARR}$ *DecValue Integer*

This command will change the entry value at the index specified by *DecValue* to the specified value. It will cause an error to occur if the index value exceeds the size of the array.

`REMOVE` { *ParamName*$_{SARR}$ | *ParamName*$_{IARR}$ } *DecValue*

This command will remove the selected index entry from the array. It will cause an error to occur if the index value exceeds the size of the array.

`TRUNCATE` { *ParamName*$_{SARR}$ | *ParamName*$_{IARR}$ } *DecValue*

This command will remove the specified number of entries from the end of the array. If the value exceeds the size of the array, it will remove all entries from the array. The number of entries can be omitted, in which case it will only remove the last entry.

`POP` { *ParamName*$_{SARR}$ | *ParamName*$_{IARR}$ } *DecValue*

This command will remove the specified number of entries from the begining of the array. If the value exceeds the size of the array, it will remove all entries from the array. The number of entries can be omitted, in which case it will only remove the first entry.

`CLEAR` *ParamName*$_{IARR}$ | *RESPONSE*

This command will remove all entries from the specified array. Note that this command will also accept RESPONSE as a parameter name.

File commands

These commands perform actions on the I/O (files or console output). Note that all file references will be relative to the directory specified by *TestPath* in the Properties file, or from the execution directory if *TestPath* is not found.

`PRINT` *`String`*

This command will output the specified text to the console (can include parameter references using $).

`DIRECTORY` *`Path`* `{` *`-f`* `|` *`-d`* `}`

This command will return the directory information of *`Path`* (relative to the TestPath) in `$RESPONSE`. Optionally allows *`-f`* to only return files or *`-d`* to only return directories. Default is to return both.

`FEXISTS` *`Filename`* `{` *`EXISTS`* `|` *`READABLE`* `|` *`WRITABLE`* `|` *`DIRECTORY`* `}`

This command will test whether the specified filename exists and will return a 1 if so and a 0 if not in the `$RESULT` parameter. Note that this would be converted to TRUE and FALSE and used by Boolean conditions. Optionally can specify whether to test whether the entry exists, or if it is a readable file, a writable file, or a directory.

`FDELETE` *`Filename`*

This command will delete the specified file, if it exists. An error will be reported if the file does not exist.

`FCREATER` *`Filename`*

This command will create the specified file and open it for reading. If a file is already open or already exists, it will report an error.

`FCREATEW` *`Filename`*

This command will create the specified file and open it for writing. If a file is already open or already exists, it will report an error.

`FOPENR` *`Filename`*

This command will open the specified file for reading. If a file is already open or does not exist, it will report an error.

`FOPENW` *`Filename`*

This command will open the specified file for writing. If a file is already open or does not exist, it will report an error.

`FCLOSE` *`Filename`*

This command will close the specified file. It will report an error if the file is not currently open.

`FREAD` *`Unsigned`*

This command will read the specified number of lines from the currently opened Read file and place the contents in the `$RESPONSE` array, each index entry receiving 1 line. If there is no file currently open for reading, it will indicate an error.

`FWRITE` *`String`*

This command will append the specified String as a line to the end of the currently opened Write file. If there is no file currently open for writing, it will indicate an error.

Parameter Assignments

Sets a parameter to the specified value. The value can be a concrete entry (such as `125` or `“hello”`) or can be a parameter or a formula such as `($RESULT * 4) + 23`. Note that parameters in the formula must all be preceeded with a ‘`$`’ and care must be taken to only assign numeric entries to the numeric parameters (I_, U_ and A_). If a String parameter value is assigned, it must contain a valid numeric value, or an error will occur. This is equivalent to the Parameter Assignment statement (SET keyword is optional). The following shows some examples:

```
Profit = “5000”

I_Value = ($U_Cost + $I_Tax) * ($Profit + 20)

I_Index = 22
```

The only assignments that can be done with String parameters is assignment to a concrete value, a String parameter (or portion of a String parameter using the bracketed version), or a concatenation of either concrete or parameter values using the ‘`+`’ sign.

```
DisplayVal = “Hello”

Person = “everybody!”

Greeting = $DisplayVal + “ “ + $Person
```

Parameters

Parameters are a feature that allows data to be collected, manipulated, reported and used to allow commands to be issued with dynamically changing values. The format of all parameter names must begin with an alpha character (A-Z or a-z) and be composed of only alphanumeric characters and the underscore character. All parameters are case sensitive (Hench Param1 and param1 are distinctly different names). When defining a parameter, you simply use the name of the parameter to defined followed by an '=' sign and then the value to assign to it. When using a parameter in a command (or as the right-hand field of an assignment) you must preceed the name with a '$' character to indicate you are specifying a parameter and not a String value.

There are 6 types of parameters that can be defined:

- *Integer* – they are 64-bit signed values defined by the name starting with "`I_`"
- *Unsigned* – they are 32-bit unsigned values defined by the name starting with "`U_`"
- *Boolean* – they are TRUE or FALSE values defined by the name starting with "`B_`"
- *String* – they are defined by default, the name starts with anything other than `I_`, `U_`, `B_`, `A_` or `L_`.
- *IntArray* – they are an array of Integer types that are defined by the name starting with "`A_`"
- *StrArray* – they are an array of String types that are defined by the name starting with "`L_`"

Booleans

Booleans consist of the values *TRUE* or *FALSE* (or *true* / *false*). If an *Integer* value is assigned to it, a 0 will be converted to FALSE and a 1 will be converted to TRUE, with any other value causing an error. If a *String* value is assigned to it, it must be composed of either the values TRUE or FALSE (case insensitive) or a numeric value that will again be interpreted as FALSE for 0 and TRUE otherwise. No operations are valid for *Booleans*, but it can take a *String* or *Integer* comparison value that returns a *Boolean* result.

Examples

```
B_Value = TRUE
B_Value = $I_Value > 75
```

Integer

Integers consist of either an optional sign followed by numeric digits, or “x” or “0x” followed by hexadecimal digits. If hexadecimal digits are represented, the value is taken as an unsigned 32-bit integer value having a range of 0x00000000 to 0xFFFFFFFF. *String* parameters can be assigned to an *Integer* parameter only if they follow the previous rule mentioned on the requirements for an *Integer* value. *Boolean* parameters can be assigned to an *Integer* parameter where a TRUE value will be converted to a 1 and FALSE to a 0. The following operations are allowed for *Integers*: { `+`, `-`, `*`, `/`, `%` }.

Operations

The operations for *Integers* work as follows:

+	Addition
-	Subtraction
*	Multiplication
/	Integer division (truncates result)
%	Modulus (get remainder of integer division

The operators are used as `Parameter = Value1 OP Value2` , where `Parameter` is the parameter being modified, `OP` is the specified operation and `Value` can either be an *Integer* or a Parameter having an *Integer* value. The equation can consist of multiple operations as well as parenthesis for forcing the order they are performed in. The calculation uses the standard order of operations: Parenthesis first, followed by math, division and modulus (from left to right), followed by addition and subtraction (from left to right).

Examples

```
I_Value = 127 * $I_Mult
I_FlagBits = 0x007F
```

Unsigned

Unsigned parameters consist of either numeric digits, or “x” or “0x” followed by hexadecimal digits. If hexadecimal digits are represented, the value is taken as an unsigned 32-bit integer value having a range of 0x00000000 to 0xFFFFFFFF. *Integer* values can be assigned to an *Unsigned* parameter, but will be truncated to 32-bits and treated as an unsigned value. *String* parameters can be assigned to an *Unsigned* parameter only if they follow the previous rule mentioned on the requirements for an *Unsigned* value. *Boolean* parameters can be assigned to an *Unsigned* parameter where a TRUE value will be converted to a 1 and FALSE to a 0. The following operations are allowed for *Unsigned*: { **+**, **-**, *****, **/**, **%** } and the following additional bitwise operators: { **AND**, **OR**, **XOR**, **ROL**, **ROR** } and the ‘**!**’ char can be placed in front of either a parameter, a numeric value, or a parenthesized block to invert the result.

Operations

The operations for *Unsigned* are the same as for *Integers* as listed above.

The additional bitwise operators are used as follows (Note: the value must be an unsigned 32-bit value and the result of the operation will always keep the result as an unsigned 32-bit value):

!	Invert all 1’s and 0’s of the value following it
AND ***value***	bitwise AND with ***value***
OR ***value***	bitwise OR with ***value***
XOR ***value***	bitwise exclusive-OR with ***value***
ROL ***bits***	Rotate bits left by specified amount of ***bits*** (for N = 1 to 31, bit_N gets bit_{N-1} and bit_0 gets bit_{31} value)
ROR ***bits***	Rotate bits right by specified amount of ***bits*** (for N = 0 to 30, bit_N gets bit_{N+1} and bit_{31} gets bit_0 value)

Examples

```
I_Value = 127 * $I_Mult
I_FlagBits XOR= 0x007F
```

String

Strings consist of any ASCII printable character (values 0x20 – 0x7E), which may be enclosed in double quotes. If the String is to consist of multiple words, that is, some spaces will be included in it, you MUST enclose the String with quotes. If it is a single word, the quotes are optional. *Integers* and *Booleans* can be assigned to a *String* parameter (the Boolean value will be converted to “TRUE” or “FALSE”). Operations that are allowed for *Strings*: { +, CONCAT, TRUNCL, TRUNCR, LENGTH, UPPER, LOWER, SUB }.

Operations

The operations allowed for *Strings* are as follows:

+ ***StrVal***	Concatenate StrVal to the right of the current *String* value

Extensions

Extensions are additions that can be placed on the end of a parameter on the right side of a calculation, assignment, or comparison.

.UPPER	Converts the *String* to all uppercase characters
.LOWER	Converts the *String* to all lowercase characters
.SIZE	returns *Integer* length the *String*
.ISEMPTY	returns *Boolean* **TRUE** if the the *String* is empty, **FALSE** otherwise

Examples

```
MyString = First                       MyString is now: First
MyString = $MyString + Second          MyString is now: FirstSecond
MyString = $MyString.TRUNCL 2          MyString is now: rstSecond
MyString = $MyString.UPPER             MyString is now: RSTSECOND
I_Length = $MyString.LENGTH            I_Lemgth is now: 9
```

IntArray

IntArrays consist of a comma series of 0 or more *Integer* values enclosed in braces. You can access (read or modify) any entry in the *IntArray* using the square brackets with the numeric index (starting at 0), such as **`I_Value = $A_Table[6]`** to get the 7th value in the array and assign it to the parameter I_Value. If an assignment is done to an *Integer* or the use in a command requires a *Integer* instead of an *IntArray* and the parameter is an *IntArray* type, only the 1st element will be used from the *IntArray* (element [0]).

Operations

These are commands that are only supported by the array type parameters. They are used like the other program commands in that the command comes first, followed by the parameter name it applies to, followed by any other additional parameters.

INSERT *`ParamName Integer`*	inserts `Integer` at beginning of `ParamName` array
APPEND *`ParamName Integer`*	appends `Integer` to end of `ParamName` array
MODIFY *`ParamName Index Integer`*	modifies the `Index` entry in `ParamName` array to the value of `Integer`
REMOVE *`ParamName Index`*	removes the `Index` entry from `ParamName` array
TRUNCATE *`ParamName Integer`*	removes `Integer` entries from the end of `ParamName` array (default to 1 entry)
POP *`ParamName Integer`*	removes `Integer` entries from the start of `ParamName` array (default to 1 entry)

Extensions

Extensions are additions that can be placed on the end of a parameter on the right side of a calculation, assignment, or comparison.

The extensions for *IntArrays* work as follows:

[`index`]	returns *Integer* value of the specified index entry
.SIZE	returns *Integer* number of entries in the *IntArray*
.ISEMPTY	returns *Boolean* **`TRUE`** if the the *IntArray* is empty, **`FALSE`** otherwise

Examples

```
A_Table1 = { 22, 25, 27, 29 }          A_Table1 is now: 22 25 27 29
A_Table2 = $I_Values 16                A_Table2 is now: 10 16
A_Table1.REMOVE                        A_Table1 is now: 22 25 27
A_Table2.PUSH $RESPONSE                A_Table2 is now: 29 10 16
```

StrArray

StrArrays consist of a comma-separated series of 0 or more *String* formatted words enclosed in braces. You can access (read or modify) any entry in the *StrArray* using the square brackets with the numeric index (starting at 0), such as **`StrValue = $L_Table[3]`** to get the 4h value in the *StrArray* and assign it to the parameter StrValue. The individual *String* element returned from a [x] nomenclature follows the same rules as any other *String*. If an assignment is done to a *String* or the use in a command requires a *String* instead of a *StrArray* and the parameter is a *StrArray* type, only the 1st element will be used from the *StrArray* (element [0]). If the command requires or allows a *StrArray* type, using the *StrArray* name will supply the entire list of entries. The valid operations for *StrArrays* are: { ADD, REMOVE, PUSH, POP, SIZE, ISEMPTY, LIST }.

IntArrays and *StrArrays* can also be assigned to a *Boolean* parameter using the element selection [x] to identify which element is being referred to (or will default to the 1st element).

When parameters are referenced in an assignment to a parameter or in a command as a replacement for a value, they must be preceeded by the '$' character to indicate this. If the parameter is not found and the assignment is to a *String* or *StrArray* parameter, the value will be accepted at face value. That is, if the command **`StrParam = $NotFound`** is used and **`NotFound`** is not a defined parameter, the value of StrParam will be "$NotFound".

Operations

These are commands that are only supported by the array type parameters. They are used like the other program commands in that the command comes first, followed by the parameter name it applies to, followed by any other additional parameters.

INSERT *`ParamName String`*	inserts `String` at beginning of `ParamName` array
APPEND *`ParamName String`*	appends `String` to end of `ParamName` array
MODIFY *`ParamName Index String`*	modifies the `Index` entry in `ParamName` array to the value of `String`
REMOVE *`ParamName Index`*	removes the `Index` entry from `ParamName` array
TRUNCATE *`ParamName Integer`*	removes `Integer` entries from the end of `ParamName` array (default to 1 entry)
POP *`ParamName Integer`*	removes `Integer` entries from the start of `ParamName` array (default to 1 entry)

Extensions

Extensions are additions that can be placed on the end of a parameter on the right side of a calculation, assignment, or comparison.

The operations for *StrArrays* work as follows:

[`index`]	returns *String* value of the specified index entry
.SORT	returns the StrArray alphabetically sorted (note: A-Z comes before a-z)
.REVERSE	returns the StrArray in reverse order
.SIZE	returns *Integer* number of entries in the *StrArray*
.ISEMPTY	returns *Boolean* **`TRUE`** if the the *StrArray* is empty, **`FALSE`** otherwise

Examples

`L_Table1 = { This is a StrArray sample }`	*`L_Table1 is now: This is a StrArray sample`*
`L_Table2 = Silly`	*`L_Table2 is now: Silly`*
`MODIFY L_Table1 3 $L_Table2`	*`L_Table1 is now: This is a Silly sample`*
`POP  L_Table2`	*`L_Table2 is now:`*
`B_Status = $L_Table2.ISEMPTY`	*`B_Status is now: TRUE`*

Parameter Conversions

Boolean

Boolean parameters must be assigned a numeric value

Integer

Integer parameters must be assigned a numeric value

Unsigned

Unsigned parameters must be assigned a numeric value

String

String parameters must be assigned a numeric value

IntArray

IntArray parameters must be assigned a numeric value

StrArray

StrArray parameters must be assigned a numeric value