# 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
```

Finally, it can also be run in server mode where it communicates with an external program (such as ScriptLauncher) that will issue commands to load, compile and run or step through a script and receive reports back that communicates information about the program status. This allows a script to be more easily tested. Refer to the Script Launcher User Guide for additional details about the interface and how to test a script.

## *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 the program and will be placed in the directory you execute from (NOT the location of the jar file). 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
DebugFileOut=debug.log
MsgEnable=0x2F
PdfPath=/home/dan/Records/Finance/Credit_card_statements/2025/Chase_VISA_3996
SpreadsheetPath=/home/dan/Records/Finance/Amazon/Testing
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 `TestPath` and `TestFileOut` values are only used in Program mode and `DebugFileOut` is only used in GUI mode.

**`TestPath`**

Defines the base path used for locating the script file to run, the `TestFileOut` location, and initial path used by the File Commands. If not defined, it will use the current directory.
(set by the `TESTPATH` program command in the STARTUP section)

**`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 `LOGFILE` program command in the STARTUP section)

**`MsgEnable`**

Selects the debug messages that are enabled
(set by the `-debug` option, the `LOGFILE` program command, and the GUI panel message checkboxes)

**`PdfPath`**

Defines the path to read the PDF file from. If not defined, it will use the current directory.
(set by the `-ppath` and `-pfile` options 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 `-spath` and `-sfile` options 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.

**`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 name is referenced to the value of `SpreadsheetPath` when a relative path (or no path) is included in the filename. If the filename is omitted, output will go to standard out.
(not set by any command, must be set manually)

## *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 }$_{1\text{-}N}$ | (whitespace) |
| Underscore | { _ } | can be used in variable names (not 1st character) |
| Dash | { - } | used in negative Integer values and for performing subtraction |
| Comma | { , } | used to separate entries in array lists |
| DblQuote | { " } | used to enclose Strings containing space characters |
| LParen | { ( } | used to start a priority operation |
| RParen | { ) } | ends a priority operation |
| LBracket | { [ } | used to start an index value or range for String or Array vars |
| RBracket | { ] } | ends the index value or range |
| MathOp | { + \| - \| * \| / \| % } | allowed math operations |
| BitOp | { AND \| OR \| XOR \| NOT \| ROR \| ROL } | allowed logical operations |
| CompSign | { == \| != \| > \| < \| >= \| <= } | allowed comparison operations |
| DecDigit | { 0-9 } | |
| HexDigit | { 0-9 \| A-F \| a-f } | |
| LowerAlpha | { a-z } | |
| UpperAlpha | { A-Z } | |
| Alpha | {UpperAlpha \| LowerAlpha} | |
| NumericHex | 0x {HexDigit}$_{1\text{-}8}$ | |
| NumericDec | - {DecDigit}$_{1\text{-}10}$ | |
| NumericLong | - {DecDigit}$_{1\text{-}19}$ | |
| UnquotedStr | {Printable}$_{1\text{-}N}$ | |
| QuotedStr | " {Printable \| WS}$_{0\text{-}N}$ " | |
| VarType | { Integer \| Unsigned \| Boolean \| String \| IntArray \| StrArray } | |
| Trait | { UPPER \| LOWER \| FILTER \| SIZE \| ISEMPTY \| SORT \| REVERSE \| DOW \| DOM \| DOY \| MOY \| DAY \| MONTH } | |
| VarReserved | { RESPONSE \| STATUS \| RANDOM \| TIME \| DATE } | |

Simple Data type definitions

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

Multi-element Data type definitions (whitespace is ignored between elements)

| | |
|---|---|
| IntArray | { Integer { , Integer }$_{0\text{-}N}$ } |

```
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)
VarName         {Alpha} {Alpha | DecDigit | _ }0-19
```

## Right-side entity definitions

```
ParamValue      {Integer | Unsigned | Boolean | String | IntArray | StrArray}
VarRef          $ VarName       (includes VarReserved values)
                $ VarNameSTR [  NumericDec [ - NumericDec ] ]            (returns partial String)
                $ VarNameSARR [  NumericDec ]                            (returns String entry)
                $ VarNameIARR [  NumericDec ]                            (returns Integer entry)
                $ { VarNameSTR | VarNameSARR | VarNameIARR }.Trait       (returns Trait value)
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    ParamNameINT = Calculation
                SET    ParamNameSTR =  String +  String +  String …
                SET    ParamNameBOOL =  Calculation CompSign 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 both discrete data values and variable references can be used as arguments for these commands, but not Calculations or String Concatenation entries. Those can only be used in the Program commands.

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

## Setup Commands

These commands only setup values that will be used by other commands – they don't have any other action associated with them.

`-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 |
| COMPILE | 0x0040 | 64 | Dk Violet | Normal | Compiler messages |
| VARS | 0x0080 | 128 | Dk Violet | Normal | Variable assignment 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
```

`-spath S_path`

Sets the location of where the spreadsheet files will be accessed from. This path value is saved in the Properties File as `SpreadsheetPath`.

`-ppath S_path`

Sets the location of where the PDF files will be read from. This path value is saved in the Properties File as `PdfPath`.

## Execution Commands

`-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.

`-snew S_file S_tabname L_array`

Creates a new Spreadsheet file (name must be an .ods file and must not currently exist) with the specified tab name and with column names defined by the `L_array` entries. The number of valid columns will be set by the number of entries in the array, while the number of rows will be set to 1. Additional entries should be appended by the `-sadd` command. The file will be saved to the `SpreadsheetPath` entry in the Properties file entry if a pathname is not given. Absolute path can be included in the name, in which case it will follow the same rules as the `-sfile` command.

`-tab U_tab | S_tab`

Sets the tab of the current Spreadsheet to use in operations. This can either be the index, starting at 0, or the name of the tab sheet.

`-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*.

`-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*

`-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 consists 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
- Variable 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.

### *Pre-Compiler Commands*

These commands are executed prior to the Compiler running, so only simple (discrete value) parameters are allowed, other than allowing the use of `<$SCRIPTFILE>` to be substituted with the base name of the scriptfile (excluding the path and file extension) to allow ease of defining the logfile name to automatically match the script file name.

`STARTUP`

Begins the section of commands that set up the environment for the script and is optional. If included, this section must be the first commands in the script – no other command should occur prior to this section. If included, it only allows the following commands to be executed:

`LOGFILE` – to specify the message filter and the file to send the debug messages to
`TESTPATH` – to set the base directory for the test scripts
`ENDSTARTUP` – to end the STARTUP section

`TESTPATH` *`PathName`*

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 log file is referenced from. By default the current user directory you are executing from will be used. This path value is saved in the Properties File as `TestPath`.

`LOGFILE` *`debugFilter bAppend fileName`*

`LOGFILE` *`debugFilter`*

Sets the debug message filter and the file to output the debug messages to. The bAppend argument is a Boolean that is set to `TRUE` if you want to append to an existing file and `FALSE` to always create a new file (default is `FALSE` if omitted). The debug filter is the same as for the -debug command option values. If filename argument is omitted, all messages will go to stdout.

`ENDSTARTUP`

Ends the setion of commands that will be run during pre-compiler mode. After this, the remaining script will be parsed for commands that will first be compiled and then executed.

The reset of these commands can be located anywhere in the script file, but will be executed on this first pass of a compile in order to extract the variable allocations defined so they can be recognized on the normal (second pass) of the Compiler.

`ALLOCATE` *`Scope VarType VarName`*

`ALLOCATE` *`Scope VarType`* { *$VarName_0$ , … , $VarName_N$* }

Defines one or more variables to be used in the script. If more than one Variable is defined on the line the names should be comma separated and enclosed in braces. All variables must be allocated before they can be used. Note that the variable type will apply to all the variables for that command. If multiple types of variables are needed, do each on a separate line. *Integer* and *Unsigned* types are initialized to 0, *Boolean* types to FALSE, *Strings* to an empty String, and *Array* types to an empty array. The Scope of the variables must be specified to be either `GLOBAL`

or LOCAL. A GLOBAL variable will be accessible to the entire script, whereas a LOCAL will be confined to the routine in which it is defined. This allows subroutines to reuse the same variable names if they are all kept as LOCALs. If there are no subroutines in the script, it doesn't make any difference whether GLOBAL or LOCAL.

```
Examples:
        ALLOCATE GLOBAL String Var1
        ALLOCATE LOCAL Integer { I_Counter, Index }
```

### SET *VarName* = { *Integer* | *Calculation* }

The SET keyword is optional in this command. 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 since they are variable *references* (being read rather than written to).

```
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 }
```

### EXIT

This can be placed at any point in the Main function of the script to terminate execution.

### ENDMAIN

This indicates the end of the Main function in the script and allows the user to add subroutines after this point. The script will terminate when this command is executed. This should be placed at the end of all scripts.

### *Subroutine commands*

`SUB` *`SubroutineName`*

This defines the start of a subroutine. The end will be marked with an `ENDSUB`. At any point (and more than once) within this you may perform a `RETURN` to exit to the calling function with or without a return value.

`ENDSUB`

This indicates the end of the subroutine function last defined. This command is a marker of where the subroutine ends so additional subroutines can be defined. If it is executed, it operates as a `RETURN` statement with no return value. This must be placed at the end of each subroutine.

`GOSUB` *`SubroutineName`*

This calls the specified subroutine passing any optional number and type of arguments. The program will begin executing from the start of the subroutine, and upon a `RETURN` will pick up on the instruction following this `GOSUB` command.

`RETURN` *`ReturnValue`*

This should be used at least once in each subroutine. It returns control to the line following the `GOSUB` that called the function. A *String* response value can optionally be passed back to the caller, which can be referenced using `$RETVAL`. If no return value is defined, `$RETVAL` will return an empty string.

### *Conditional commands*

```
IF { NOT | ! } { VarName | Comparison | Boolean }        (boolean comparison)
IF { VarName } CompSign Calculation                      (numeric comparison)
IF { VarName } CompSign { VarName | String | StrArray }  (string comparison)
```

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: { `+, -, *, /, %` }. There are 3 forms of comparison allowed:

- Boolean – This can have the following forms, each of which may be preceeded by *`NOT`* or *`!`* :
  - single discreet value (`TRUE` or `FALSE`)
  - single Boolean Variable
  - Boolean Variable compared to (`TRUE` or `FALSE`) with either `==` or `!=`
  - Boolean Variable compared to another Boolean Variable with either `==` or `!=`
  - Comparison of a Numeric Variable to a Numeric Calculation ( e.g. `$NumValue1 > $NumValue2 + 3` )
- Numeric – This can have the following forms:
  - Numeric Variable compared to discrete numeric integer
  - Numeric Variable compared to another Numeric Integer
  - Numeric Variable compared to Numeric Calculation ( e.g. `5 x ($Value + 12)` )
- String – This can have the following forms:
  - String Variable compared to discrete String value (use quotes if multiple words)
  - String Variable compared to another String value
  - String Variable compared to a StrArray value (compared to each element of it)

When a String is compared to a StrArray (either discrete list or a StrArray Variable), it will be compared to each element of the array and the conditional result will be `TRUE` if any of the values matched. Note that in this case, only the == or != is allowed for the comparison. Otherwise, String comparisons will do alphabetical comparisons of the elements, where ‘z’ is interpreted as greater than ‘a’. If the two strings match, but one has additional characters in it, it will be considered the larger value.

If the condition was met (conditional result was `TRUE`), the code following the IF statement will be executed until it encounters an `ELSE`, `ELSEIF` or `ENDIF` statement, which will cause it to jump to the line following the `ENDIF` statement. If the condition was not met, it will proceed to the next `ELSE`, `ELSEIF` or `ENDIF` statement. IF statements are allowed to 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 not TRUE
       IF $L_STRArray.ISEMPTY        ## checks if the String array is empty (0 elements)
       IF $StrValue == { Mon, Tue, Wed } ## checks StrValue  against all 3 values
```

### `ELSEIF` *`VarName 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.

### *Loop commands*

FOR *VarName = InitValue { TO | UPTO } EndValue STEP StepValue*

*FOR EVER MaxLoops*

This allows a series of commands to be repeated for a specified number of times. The format of the arguments is defining the loop variable to use, which can be accessed within the loop as a variable reference. This name follows the same format as other variables (startes with a letter and consists of a combination of letters, decimal numerics, and the underscore character. It must be unique and not conflict with any user variables or reserved variables/commands, but can share the name of another loop as long as one is not contained within the other. The first argument in the command is the loop variable name followed by an equals sign (optional) 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. Next is either the *TO* or *UPTO* string followed by the *EndValue* for the loop variable, which is the value of the loop variable that will cause the loop to terminated. If the *TO* value is used, the loop will execute the *EndValue* value before terminating, an the *UPTO* value will terminate without executing the *EndValue*. The last optional arguments specify the *StepValue*, which is the amount to add to the loop variable on each iteration. This value can be a positive or negative value and can also be a variable reference. If omitted, the step size is assumed to be +1. Loops can also be nested, as long as the loop variable names are not the same. Note that the loop will not run at all if the *InitValue* exceeds the conditions of the *EndValue*.

The other format of the command allows you to not specify a loop parameter or a terminating condition. The *EVER* will indicate looping until some BREAKIF statement (or an IF statement that has BREAK as its action statement) will break the loop. This effectively allows creating Do-While or Do-Until loops that have termination conditions that are not easily handled by simply running a certain number of loops. Care must be taken when using this to guarantee that the loop will be executed. As a safety feature, you can include an optional *MaxLoops* value that will limit the maximum number of loops that will run before exiting the loop.

```
Examples:
        FOR I_Index = 1 TO 10 STEP 1  ## will perform 10 iterations (UPTO would perform 9)
            IF $I_Count >= $I_Index
                PRINT $I_Count        ## if I_Count >= I_Index, will print value of I_Count
                BREAK                 ##  then will go to line following NEXT 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
        SET NewVal = 3                ## subsequent statement following loop
```

BREAK

This is used within the most recent FOR loop and will cause the program to jump to the line following the next NEXT statement.

SKIP

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 variable by the *StepValue* and then test to see if the *EndValue* condition has been met. If so, it will jump to the line following the NEXT statement, otherwise it will jump back to the beginning of the loop.

BREAKIF { *VarName* } *CompSign Calculation*

This is used within the most recent FOR loop and will cause the program to jump to the line following the next NEXT statement if the specified Condition is TRUE. It is the same as enclosing the BREAK command in an IF … ENDIF wrapper.

`SKIPIF` { *VarName* } *CompSign Calculation*

This is used within the most recent `FOR` loop and will cause the program to jump to the following `NEXT` commandt if the specified Condition is TRUE. It is the same as enclosing the `CONTINUE` command in an `IF … ENDIF` wrapper.

`NEXT`

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

### *Array commands*

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

`INSERT` *VarName$_{SARR}$* { *String* | *StrArray* }

`INSERT` *VarName$_{IARR}$* { *Integer* | *Calculation* | *IntArray* }

This command will insert the specified value at the beginning of the array. This command allows either a single value to be added or an array of numerics or strings.

`APPEND` *VarName$_{SARR}$* { *String* | *StrArray* }

`APPEND` *VarName$_{IARR}$* { *Integer* | *Calculation* | *IntArray* }

This command will append the specified value at the end of the array. This command allows either a single value to be added or an array of numerics or strings.

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

`MODIFY` *VarName$_{IARR}$ 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` { *VarName$_{SARR}$* | *VarName$_{IARR}$* } *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` { *VarName$_{SARR}$* | *VarName$_{IARR}$* } *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` { *VarName$_{SARR}$* | *VarName$_{IARR}$* } *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` { *VarName$_{SARR}$* | *VarName$_{IARR}$* | *RESPONSE* }

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

`FILTER` *VarName$_{SARR}$ Filter* { *!, !LEFT, !RIGHT, LEFT, RIGHT* }

`FILTER` *VarName$_{IARR}$ CompSign Value*

`FILTER` *RESET*

This command will set a filter to eliminate entries from being listed when the .FILTER extension is added to the variable name. The filter is stackable in that multiple passes of filters can be applied to filter the results. Using the RESET will reset the filter to not eliminate anything. For String Arrays, you can search for onlt entries that have a *Filter* string contained in them. The optional argument allows only matching the first (*LEFT*) or the last (*RIGHT*) characters of the string. Also a *!* char can be used to block the selected entries instead of passing them. For Integer Arrays it works the same except it is only passing values that match the numeric comparison. Note that if you have

2 arrays of the same length, they can use the same filter, allowing you to have 2 arrays that are associated with each other (such as item description and cost columns of a spreadsheet) and filter both arrays with the same data.

### *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* from the Properties file, or from the execution directory if *TestPath* is not found. Also note that the directory paths are limited to the users home directory, to prevent the possibility of contaminating the system files or other users directories. Upon startup, the default directory path is the *TestPath* from the Properties file where all file commands will operate from when specifying relative paths in the *Filename*. The CD command will set this to whatever the user selects (within the users home directory path), so that the File commands will then use relative paths based on that value. The current path can be found using the Reserved variable $CURDIR.

PRINT *String*

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

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

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.

CD *Path*

This command will set the path used by the rest of the File commands. The path can be specified as absolute (starting with a '/' character) or relative to the current path. The '~' character can also be used in place of the '/' to represent an absolute path based on the users home directory. The path specification can also consist of a ".." to bump up to the parent directory of the current one, and can use a series of them to repeat this. For instance, CD ../../.. will cause a current path of "/home/user/Tests/dira/files/hello/world" to become "/home/user/Tests/dira".

MKDIR *DirName*

This command will create the specified directory in the current path.

RMDIR *DirName*

This command will delete the specified directory and all files and subdirectories it contains from the current path.

FEXISTS { *-x* |*-r* | *-w* | *-d* } *Filename*

This command will test whether the specified file or directory exists and will return a TRUE if so and a FALSE if not in the $STATUS parameter. You can specify whether to test whether the entry is a readable (*-r*) or writable (*-w*) file, or a directory (*-d*). By default, it simply tests if the file or directory exists (*-x*).

FDELETE *Filename*

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

FCREATE { *-r* | *-w* } *Filename*

This command will create the specified file and open it for reading or writing. If the *-r* (read) or *-w* (write) is omitted, it will assume *-r*. If a file is already open or already exists, it will report an error. Only 1 file can be open for reading or writing at a time.

FOPEN { *-r* | *-w* } *Filename*

This command will open the specified file for reading or writing. If the *-r* (read) or *-w* (write) is omitted, it will assume *-r*. If a file is already open or does not exist, it will report an error. Only 1 file can be open for reading or writing at a time.

`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 the number of lines is omitted, it will read the entire file. 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.

`FGETSIZE` *`Filename`*

This command will return the size of the specified file (number of characters it contains). The value will be placed in the $RESPONSE reserved variable.

`FGETLINES` *`Filename`*

This command will return the number of lines in the specified file. It will do this by opening the file and reading the file line at a time, counting the lines. The value will be placed in the $RESPONSE reserved variable.

`OCRSCAN` *`Filename`*

This command will read the specified PDF file and scan it for text. The output will be available as the String reserved variable `$OCRTEXT`.

### *Variable Assignments*

When you mak a variable assignment, it has the same format as the SET command without using SET. This is used to set a variable to the specified value. The value can be a concrete entry (such as `125` or `“hello”`) or can be another variable or a formula such as `($RESULT * 4) + 23`. Note that variable references in the formula must all be preceeded with a ‘`$`’ and care must be taken to only assign numeric entries to the numeric variables. If a String value is assigned to a numeric variable, it must contain a valid numeric value, or an error will occur. 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 variables is assignment to a concrete value, a String variable (or portion of a String variable using the bracketed version), or a concatenation of either concrete or parameter values using the ‘`+`’ sign.

```
DisplayVal = “Hello”

Person = “everybody!”

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

## *Loops*

## *Subroutines and LOCAL Variables*

## Variables

Variables 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 variable names must begin with an alpha character (A-Z or a-z) and be composed of only alphanumeric characters and the underscore character. All variables are case sensitive (hence, Param1 and param1 are distinctly different names).

You must allocate all variables that are to be used in the program using the `ALLOCATE` command. This allocates the storage for the variable and sets the data type and its scope. The data types are defined below and the scopes allowed are: `GLOBAL` and `LOCAL`. The scope pertains to the accessibility of variable to other routines, if a program contains subroutines. If there are no subroutines, the scope is irrelevant. `GLOBAL` variables are accessible throughout the script, whether that are defined by the MAIN function or a subroutine. `LOCAL` variables are only accessible to the MAIN or subroutine in which they are allocated. Defining subroutine parameters as `LOCAL` allows you to reuse the name in other routines, since they are only visible to the one function. This also prevents other functions from accidentally changing the value of the variables in another function.

When assigning a value to a variable, you simply use the name of the variable to defined followed by an '=' sign and then the value to assign to it. When referencing a variable value in a command (or as the right-hand field of an assignment) you must preceed the name with a '$' character to indicate you are specifying a variable reference and not a String value. If the reference is embedded in a String (no spaces around it) you can use the following syntax to have it expanded upon execution: `XXX\$VarName$XXX`.

There are 6 variable data types that can be defined:

- *Integer* – are 64-bit signed values
- *Unsigned* – are 32-bit unsigned values
- *Boolean* – are TRUE or FALSE values
- *String* – are ASCII strings (if spaces are to be included, you must enclose the value in "quotes")
- *IntArray* – are an array of Integer types
- *StrArray* – are an array of String types

### *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. A *String* variable can be assigned to an *Integer* variable only if it follows the previous rule mentioned on the requirements for an *Integer* value. *Boolean* variables can be assigned to an *Integer* variable 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 variable being modified, **`OP`** is the specified operation and **`Value`** can either be a discreet *Integer* value or a Variable 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).

### *Extensions*

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

.HEX Converts the numeric value to a hexadecimal *String*

### *Examples*

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

## *Unsigned*

*Unsigned* variables 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* variable, but will be truncated to 32-bits and treated as an unsigned value. *String* variables can be assigned to an *Unsigned* variable only if they follow the previous rule mentioned on the requirements for an *Unsigned* value. *Boolean* variables can be assigned to an *Unsigned* variable 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 variable, 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) |

### *Extensions*

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

| | |
|---|---|
| .HEX | Converts the numeric value to a hexadecimal *String* |

### *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* variable (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 variable on the right side of a calculation, assignment, or comparison.

| | |
|---|---|
| [ `index` ] | returns the specified characters from the string variable |
| .UPPER | Converts the *String* to all uppercase characters |
| .LOWER | Converts the *String* to all lowercase characters |
| .TOLINES | Converts the *String* to *StrArray* breaking it into lines (split by newline char) |
| .TOWORDS | Converts the *String* to *StrArray* breaking it into words (split by space char) |
| .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 variable 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 variable 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 variables. They are used like the other program commands in that the command comes first, followed by the variable 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 variable 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 |
| .FILTER | returns *IntArray* that has entries filtered by the *`FILTER`* command |
| .HEX | Converts the numeric values to *StrArray* of hexadecimal values |

### *Examples*

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

```
A_Table1 is now: 22 25 27 29
A_Table2 is now: 10 16
A_Table1 is now: 22 25 27
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 variable 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 variable 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* variable using the element selection [x] to identify which element is being referred to (or will default to the 1st element). If a *StrArray* variable consists of numeric values, it can be used as an *IntArray* reference and the entries will be converted over.

When parameters are referenced in an assignment to a variable or in a command as a replacement for a value, they must be preceeded by the '$' character to indicate this. If the variable 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 variable. They are used like the other program commands in that the command comes first, followed by the variable 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 |
| .FILTER | returns *StrArray* that has entries filtered by the *`FILTER`* command |

### *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
```

## Extensions (Traits and Bracketing)

Variables may have extensions attached to the by way of either Bracketing (to select an individual element or range of elements) or Traits (to get specific attributes of the variable). The following list the allowed types, the type of variables they are allowed for, and what they do.

These are for all types:

| | |
|---|---|
| .WRITER | returns the script line that performed the last write to the variable |
| .WRITETIME | returns the timestamp (referenced from start of script) of the last write to the variable |

These are for Integer and Unsigned types:

| | |
|---|---|
| .HEX | Converts the numeric value to a hexadecimal *String* |

These are for String types:

| | |
|---|---|
| [ `index` ] | returns the specified characters from the string variable |
| .UPPER | Converts the *String* to all uppercase characters |
| .LOWER | Converts the *String* to all lowercase characters |
| .TOLINES | Converts the *String* to *StrArray* breaking it into lines (split by newline char) |
| .TOWORDS | Converts the *String* to *StrArray* breaking it into words (split by space char) |
| .SIZE | returns *Integer* length the *String* |
| .ISEMPTY | returns *Boolean* **`TRUE`** if the the *String* is empty, **`FALSE`** otherwise |

These are for IntArray types:

| | |
|---|---|
| [ `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 |
| .FILTER | returns *IntArray* that has entries filtered by the *`FILTER`* command |
| .HEX | Converts the numeric values to *StrArray* of hexadecimal values |

These are for StrArray types:

| | |
|---|---|
| [ `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 |
| .FILTER | returns *StrArray* that has entries filtered by the *`FILTER`* command |

These are only to be used for the `$DATE` reserved variable:

| | |
|---|---|
| .DOW | returns the *Integer* day of the week |
| .DOM | returns the *Integer* day of the month |
| .DOY | returns the *Integer* day of the year |
| .MOW | returns the *Integer* month of the year |
| .DAY | returns the *String* day of the week |
| .MONTH | returns the *String* month of the year |

## Reserved Variables

There are several pre-defined variables that can be used on the right-side of equations or as arguments to commands. These give access to some common information that can be useful:

### *$RESPONSE*

This is defined as a StrArray type and is set from several of the commands as a way of returning String and multi-string response values to the script. The commands that do this are:

| | | |
|---|---|---|
| `DIRECTORY` | `multi-entry` | returns the directory contents |
| `-clip` | `multi-entry` | returns the clipboard file lines (one line per array entry) |
| `-pfile` | `multi-entry` | returns the PDF file lines (one line per array entry) |
| `-date` | `1 entry` | returns the formatted date (assumes future date if a relative day was given) |
| `-datep` | `1 entry` | returns the formatted date (assumes past date if a relative day was given) |
| `-maxcol` | `1 entry` | returns the max column size of spreadsheet loaded |
| `-maxrow` | `1 entry` | returns the max row size of spreadsheet loaded |
| `-find` | `1 entry` | returns the row of the item number found in spreadsheet |
| `-class` | `1 entry` | returns the |
| `-cellget` | `1 entry` | returns the currents cell contents) |
| `-cellclr` | `1 entry` | returns the previous cell contents before clearing entry |
| `-cellput` | `1 entry` | returns the previous cell contents before writing new value |
| `-colget` | `multi-entry` | returns the contents of the column read |
| `-rowget` | `multi-entry` | returns the contents of the row read |

It allows the same options as the StrArray parameters – that is, you can use [brackets] to select a specific String entry and the following Extensions are allowed:

```
.SIZE
.ISEMPTY
```

Note that the values are always appended to the current array contents. If you want only the data from a single command, you must first clear the $RESPONSE netry before issuing the command. This parameter can be cleared with the command:

```
CLEAR $RESPONSE
```

### *$RETVAL*

This is defined as a String type and is set by the last subroutine called if it specifies a value in its RETURN command.

### *$STATUS*

This is defined as a Boolean type and is set by the following commands:

`FEXISTS` returns the the status of command (`TRUE` or `FALSE`)

### *$SCRIPTNAME*

This is defined as a String type and returns the base name of the script file that is running (i.e. it removes the directory path and the file extension. This allows using the scriptname for using the same base name for files that are created or to place in an output file to identify the script that produced it.

### *$CURDIR*

This is defined as a String type and returns the current directory path selection used by the File Commands. The value will change when the `CD` command is used to change directories.

### *$RANDOM*

This is defined as an Unsigned type and returns a random number between 0 and the MaxRandom value. The MaxRandom value is initially set to 1000000000. The value generated will be a value >= 0 and < MaxRandom. The MaxRandom value can be changed by the command:

```
RANDOM = UnsignedValue
```

### *$OCRTEXT*

This is defined as a String type and returns the text of that last scan performed by *OCRSCAN*.

### *$TIME*

This is defined as a String type and returns the current time formatted as: `HH:MM:SS.mmm`

### *$DATE*

This is defined as a String type and returns the current time formatted as: `YYYY-MM-DD`

It allows the the following Extensions:

```
.DAY          returns a String of the Day of the Week as a name (e.g. MONDAY, WEDNESDAY, etc)
.MONTH        returns a String of the Month of the Year as a name (e.g. APRIL, AUGUST, etc)
.DOW          returns an Unsigned value of Day of the Week   (range: 1 – 7, where 1 = MONDAY)
.DOM          returns an Unsigned value of Day of the Month  (range: 1 – 31Y)
.DOY          returns an Unsigned value of Day of the Year   (range: 1 – 366)
.MOY          returns an Unsigned value of Month of the Year (range: 1 – 12)
```

The following are Properties File entries that are also available for extracting the values from. They all return a String value of whatever the Properties file is set to, or an empty String if it is not defined.

### *$TestPath*

This returns the current Test path setting.

### *$PdfPath*

This returns the current Pdf path setting, where PDF files are read from.

### *$SpreadsheetPath*

This returns the current Spreadsheet path setting, where the Spreadsheets that are read and written to are located.

### *$SpreadsheetFile*

This returns the name of the last Spreadsheet file that was loaded with the `-sfile` command.

### *$SpreadsheetTab*

This returns the tab number of the Spreadsheet file that was set with the `-tab` command.

### *$TestFileOut*

This returns the name of the test message output file selected by the `-ofile` command.

### *$DebugFileOut*

This returns the name of the debug message output used by the GUI interface.

### *$MsgEnable*

This returns the value of the debug message flags that are enabled.

### *$MaxLenDescription*

This returns the value of the maximum length setting for the definitions saved to the spreadsheet from a clipboard.

## Parameter Conversions

### *Boolean*

*Boolean* variables are assigned either a `TRUE` or `FALSE` value from either a direct value or from a Boolean parameter reference. However, it will also allow a String reference variable that has either "`TRUE`" or "`FALSE`" or either an Integer or Unsigned reference variable to be assigned to it as well, with 0 representing `FALSE` and anything else as `TRUE`. It can also be assigned the result of a Comparison entry, such as

```
bValue = $intValue >= 42
```

### *Integer*

*Integer* variables are defined as signed 64-bit numeric values and can be assigned from an explicit value, a variable reference, or a Calculation. Variable references must infer a numeric value, so Strings will be automatically converted to integers as long as they are composed of numeric digits and Booleans will be converted to 0 for FALSE and 1 for TRUE. Array entries can also be assigned by specifying a single entry from the array, such as

```
intValue = $myNumArray[32]
```

Calculations allow performing a math equation and assigning the result to the variable, such as

```
iData = (52 * $RefVal) / 3
```

### *Unsigned*

*Unsigned* variables are defined as unsigned 32-bit values, and work in the same manner of assignment as the Integer type, except the data value assigned to it must be >= 0 and 32-bits or less, so values in the range x0 – xFFFFFFFF or 0 to 4294967295.

### *String*

*String* variables can take any value, but must either consist of non-space characters or must be enclosed in double quotes. Any variable reference can be assigned to it including Arrays, but it will only assign the 1st element in the array to the String variable. If a different index of the array is desired to be assigned to it, you can add the bracketing format to the array entry, such as

```
strValue = $sArray[12]
```

Strings can also be split and assigned to a StrArray using the `.TOWORDS` or `.TOLINES` Trait added to the end of the parameter.

### *IntArray*

*IntArray* variables are simply a list of Integer values that can be individually accessed using a single bracked value (such as `$intArray[12]`)or used as a whole array of numbers (or partial array by using the bracketing format with a range such as `$intArray[12-32]`). Array entries can be appended to at the beginning (using `INSERT` command) or at the end (using `APPEND`). Individual entries can be modified (`MODIFY` command) or deleted (`DELETE` command). You can also delete a specific number of entries from either the beginning of the array (`POP`) or the end (`TRUNCATE`) or delete all entries in the array using `CLEAR`.

### *StrArray*

*StrArray* variables are similar to *IntArray* types, but can carry any String value as an element.