Domain Specific Language Specification (DSL)

Draft, Version 0.6.5

PROPRIETARY & CONFIDENTIAL

# Summary

This document outlines the attributes and features of the DSL for creating Behaviors in the framework.

# Design Considerations

The design of this DSL builds upon the last 70 years of programming language design to make a language that is as simple to use as possible. This allows the DSL to be the underlying language engine for a low code/no code environment, removing barriers to entry for programmers and non-programmers alike.

In practice this means that the DSL must manage as much of the complexity present when developing software behind the scenes, allowing the programmer to focus on what they need to do and less on how to do it. To facilitate this goal, the DSL specification formalizes aspects of the language that are normally left to interpretation. For example, reading and writing to files, the web, or printer are all performed with the same keywords “read” and “write.” The programmer only needs to specify where they want the information to be read from or written to.

Another key consideration is the data transfer format. In the DSL, data is always transmitted as JSON. This allows the DSL to easily read and write data to other systems in a seamless and comprehensive manner.

The DSL is designed as a procedural language with advanced data management capabilities. Common operations like data transfer between systems is ‘built in’ to the language allowing programmers to create scripts with one step that, in other languages, would require many steps. This ability is possible because of advances in hardware, software and computer language theory.

The DSL is *not* garbage collected. In theory garbage collection allows the programmer to avoid thinking about when to delete their variables. In practice, it often becomes necessary for programmers to work around the garbage collector to keep it from freezing program resources at inappropriate times. The DSL’s approach is to automatically free memory when the variables referencing the memory go out-of-scope. This allows the programmer to have direct control over the life of its data.

# Document Conventions

The following conventions are used in this document:

|  |  |
| --- | --- |
| Expression | An expression is any statement that produces a value. For example, 3+4 is an expression because it produces a value of 7. |
| Value | Any expression that can be evaluated to produce a single value. This can be a number, function, variable, expression, or other program statement that produces a quantity. |
| Index | Any expression used to represent an element in a list or in a table. |
| Key | Any expression used to look up values in a dictionary. |
| Variable | A named storage location used by the program. Variables must be declared before they are used. The exception to this rule is global scoped variables which can be declared anywhere. Attempting to redefine a global variable will generate an error. |
| Identifier | A name for an item in a program. Identifiers begin with a letter and can only contain letters, numbers, or underscore characters. |

# Overview

The DSL language is a C-style procedural language with a few carefully chosen attributes to simplify the creation of programs for the framework. Like the C language, the number of reserved keywords in the language is kept to a minimum. A rich set of operators is supplied allowing succinct code to be easily created. The language is case sensitive so “MyName” and “myName” specify two different identifiers.

International

Program scripts are written in UTF8 compatible text format. Script identifiers are currently restricted to the standard ASCII character set to comply with international common use conventions for programming languages. This limitation does not apply to string and character values which can contain any valid UTF8 value.

# Program

A program in the DSL consists of a set of statements. Statements are executed from top to bottom of the script file in the order they are encountered. The exception to this rule is functions that are blocks of statements representing reusable code. For example, this simple one-line program represents the classic program “Hello World” in the DSL:

print (“Hello World!”);

Running the Hello World program would display the words “Hello World!” on the console.

Any code enclosed in curly braces { } is known as a block of code. Code blocks are fundamental to controlling what code is executed and organized. For example, in the program shown here:

var SayHello()

{

print (“Hello World!\n”);

}

SayHello();

Hello World! message is not displayed on the console until called by the SayHello(); line of code. The reason for this is that the SayHello() function definition defines a group of statements named SayHello that are only executed when the program explicitly says to execute them.

A program does not have to contain executable code. It is possible to define a program that simply contains data. For example:

var global AirBases = { “Keesler”, “Macdill”, “Langley” };

This defines a program that contains a collection of three Air Force base names. The collection can be accessed in other programs since the collection is defined as globally accessible. For example, if the above program were called AirForceBases then the following program would be able to display the bases as shown below:

print(AirForceBases.AirBases[0], AirForceBases.AirBases[1], AirForceBases.AirBases[2]);

The AirForceBases program containing the airBases list must be accessible to the DSL compiler. This means that the AirForceBases program needs to be compiled along with the script that is calling it.

# Statements

All lines of code begin with a statement. A statement is a special keyword the language recognizes as a command to do something. For example:

var a = 10;

The keyword “var” tells the language to create a location in the program’s memory called “a” and assign the number 10 to it. A statement usually ends with a semicolon but can end in special cases with other characters.

Keywords are commands in the DSL. They instruct the program to take specific actions, such as declaring a variable or executing various parts of the code based on some conditional expression. Like the C programming language, the DSL keeps the number of keywords to a minimum. The following are examples of valid statements in the DSL:

print(“Enter a value for a:”

var a = Input();

var b = input();

if (a == b)

{

Print(“{a} is equal to {b}.\n”);

}

else

{

Print(”{a} is not equal to {b}.\n”);

}

## Keywords

In compliance with the C standard, the DSL keeps the number of keywords to a minimum. The current keyword list is as follows:

|  |  |
| --- | --- |
| Keyword | Description |
| Continue | Go to the start of a loop immediately. |
| global | Set the scope of a variable to be accessible from any program script. |
| script | Set the scope of a variable to only be accessible in the current program script. |
| return | Exit the current function and return the value of the following expression. |
| switch | Evaluate the following expression and conditionally execute a case based on the results. |
| while | Evaluate the following expression and, if true, perform the statement block following the while. *Note: Unlike most programming languages, a loop that never exits is an error condition.* |
| print | Evaluate the following expression and display the results on the standard output device. |
| input | Pause the program execution and let the user enter some information. |
| const | Prevent the following variable from being modified. Only valid when the variable is being defined. |
| write | Write the result of the following expression to a file at the specified location. The file is extended if necessary. |
| break | Exit a loop immediately. |
| local | Set the scope of a variable to be accessible only within the defined function. |
| read | Read a file at the specified location for a specified length and return the result. |
| else | Must be used with a corresponding if-statement. If the if-statement expression evaluates to *false*, the else-statement block is executed. If and else statements can be chained together as if (cond) { } else if (cond) { }. This enables the creation of complex decision tree logic to be created. |
| case | Must be used with a corresponding switch-statement. If the result of the switch-expression matches the result of the case-expression, then the statement block in the case is executed. Unlike C and C++ the statement block associated with a case must be enclosed in {}. Unlike C and C++ the statements associated with a case do not need to end with break. |
| default | Must be used with a corresponding switch statement. If the result of a switch expression does not match any of the defined cases and default is present, the statements associated with the default case are executed. |
| end | Terminates execution of the program immediately. |
| var | Declares that the identifier following the var is a variable. The default scope is script-level. Note: The var keyword is required when declaring a variable. |
| for | Creates a counting type loop composed of three seconds, an initializer, conditional, and update section. |
| dir | Returns the path and name of all files in a folder. |
| del | Removes the file specified by the following expression. |
| brk | Stops execution of the program script and enters the debugger when encountered. |
| if | Evaluates the following expression and if the result is true executes the following statement block. |
| stop | The stop command tells the compiler to ignore all text in the current script file after the stop key word is encountered. |

# Blocks

Blocks are groups of statements defined between curly braces. Blocks are important because they allow the programmer to separate a program's functionality into specific units of work. For example, in the following code, one block is executed if the variable a is less than (<) 5 and the other block is executed if the variable a is greater than (>) 5:

if (a < 5)

{

print(“The variable a is equal to 5.”);

}

else

{

print(“The variable a is some other value than 5.”);

}

# Comments

The DSL supports the same two styles of comments provided by the C programming language: line comments and multi-line comments. Line comments begin with // and continue up to the end of the line. For example:

var engines = 10; //Number of engines in the repair shop.

Would cause the compiler to ignore all text from the // up to the end of the line. Multi-line comments begin with a /\* and end with a \*/. All text within the /\* and \*/ are ignored. For example:

/\*

This program tracks the number of new tires in the warehouse.

\*/

fun CountTires()

{

…

return numberOfTires;

}

# Values

Values represent information in the DSL. They can be numbers, characters, Boolean, strings of characters. The DSL determines the type of value based on the characters that make up the value. For example, if the value is made of up numbers and a leading + or – the DSL determines that the value is a number. If the value begins with a ‘ the DSL determines the value is a single character. The following table shows the different types of values and the characters the DSL looks at to determine the type of value the programmer wants to create.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| number | -+ [0-9]. | All number characters up to the next non-number character, which may be a single period character. If a period exists in the number, the DSL assumes it is a real or floating point number and creates a double value type to hold the number. |
| char | ‘ | A single character followed by another single quote. Escape codes can be used. See *string* formatting details. |
| Boolean | true or false | The keywords “true” or “false”. A false value is numerically the number 0, while the value true is any number but 0. This is often represented as (!0) or 0 negated. |
| string | “ | All characters up to the closing “ character are considered part of the string value. A string value may include variables or values inline with the string by enclosing them in {}. For example:  string name = “Program 100”;  string s = “The program’s name is {name}”;  Would set the string “s” equal to”The program's name is Program 100”.  Strings are stored and processed in UTF8 format supporting international text processing.  Collection values can be inserted into a string using the standard [] collection access operator between {}. For example: var a = {1, 2, 3}; var s = “ collection = {[a]}” would result in the variable “s” being set to: “collection = { 1, 2, 3 }”. |
| collection | {} | A collection of values. |

## Type Conversion

The DSL automatically converts base types from right to left when compiling the source code. For example:

var a = 12.5 + 10;

print(“This value of a is ”, a);

When this code is run, the value 22.5 is printed. This is because the value of 10 is first converted to a double value and then added to the double value of 12.5, then the result 22.5 is assigned to the “a” variable. You can force a specific conversion by including the “type” to the left of the value you want to convert. For example:

var a = 10;

a = “The variable a = “ + a;

print(a);

In this example, the value in the variable “a” is converted to a string and then appended with the string “The variable a = ”, then the result is assigned to the “a” variable. Another method of forcing a conversion is to use the (cast) operator. This operator lets you specify how you want the information converted. For example,

var a = 22.5;

var b = (int)a;

print(“The variable b = ”, b);

When the previous example is run, the number 22 is printed. This occurs because the (int) cast forces double value in variable “a” to be converted to an integer before being assigned to the variable “b”.

## String Format Detail

Strings are collections of characters treated as a single value by the DSL. Strings must be enclosed in double quote characters. For example, all the strings shown here represent valid strings in the DSL:

“Hello world”

“A”

“AAAA\tBBBB\n”

The \ character is a special character known as an “escape code”. It tells the DSL parser that the characters that follow represent a special, non-printable character code. The following table shows the supported escape codes:

|  |  |
| --- | --- |
| Escape Codes | Description |
| \\ | Inserts a single \ character. For example, the string “\\One” would create the string \\One. |
| \’ | Inserts a single ‘ character. |
| \” | Inserts a single “ character. |
| \n | Inserts a single new line character. |
| \r | Inserts a single carriage return character. |
| \t | Inserts a single tab character. |
| \b | Inserts a single backspace character. |
| \f | Inserts a single form feed character. |
| \{ | Inserts a single { character. |
| \} | Inserts a single } character. |
| \[0-9] | Converts the integer numbers following the \ into a value and inserts the code that the value represents into the string.  *All numbers are evaluated up to the next character that is not a number. For example, “\14844047” would insert the value which is the UTF8 right to left mark. Note: It is not necessary to use this escape code as UTF8 characters can be entered directly into a string.* |
| \X[0-9, A-F], \x[0-9, a-f] | Converts the hex digits (base 16) value following \ into a value and then inserts the code that the value represents into the string. |
| {variable or function call} | The {} characters when used inside a string allow a value to be inserted directly into the string. This is useful to place the contents of a variable or the result of a function call into a string value. The {} are not added to the string. To add a { or } character into a string use the \{ and \} escape characters. |

# Variables

Variables are identifiers that reference a storage location within the framework. Variables are created if they do not already exist. The “var” keyword must precede the variable name. For example:

var MyVariable = 10;

This creates a variable named myVariable and assigns it a value of 10.

A variable must be a valid identifier. This means that it must begin with either a letter, or an underscore character. After the first character the variable can be made up of any number of letters, numbers, digits, or underscores. A variable cannot be named the same as any keyword. If the variable is not assigned an initial value, it will be assigned a value of 0. If a value is assigned to the variable it must be composed of values and operations that can be pre-calculated. For example, the following sets the string type variable to the string “Hello World”,

var string = “Hello” + “ “ + “World”;

If the expression can’t be pre-calculated an error is generated. For example, the code shown here will generate an error because the assignment expression can’t be precalculated,

var string = Word1() + “” “ + Word2();

This restriction also applies to variables because the value of the variable could change when the program is run. For example,

var hello = “Hello”;

var word = “Word”;

var string = hello + “ “ + word;

Would generate an error since the compiler can’t determine the actual value of hello or world at the time the variable string is created.

# Scope

Scope determines how long a variable's life is and where that variable can be accessed. By default a function or variable is set to script level scope. This means the variable is available everywhere in the current script. This is a bit different that C and C++. In those languages variables are scoped based on where they are defined. This can be confusing for some programmers as variables can be defined and then ‘go out of scope’. For example, consider the following C function:

int SomeFunction(int a)

{

If ( a == 10 )

{

int b = 20;

}

printf(“The value of b is “, b);

}

In this example the printf() line will generate an error when the program is compiled. Even though the b variable has been defined previously it went out of scope because it was defined in code statements that belonged to the if ( a == 10 ) statement. The DSL avoids this issue by allowing the programmer to explicitly define the variables life time via its scope or defaulting it to live for the life of the current script program module.

The DSL defines 3 levels of explicit scoping:

1. Global
2. Script
3. Local

### Global

Globally scoped variables remain in memory if the program in which they were defined remains in memory. You can think of global variables as permanently available. For example:

var global myVariable1001 = 1;

print(myVariable1001);

The variable myVariable1001 is available in all scripts that make up the program even inside other functions in other scripts.

### Script

Script scope makes the variable available everywhere within the current program script. This is the default scope level for variables defined outside of any function.

In the DSL, scope determines when a variable is removed from memory. Scope restricts the area in which a variable or function can be used. By default, functions and variables are scoped to the file in which they are defined. This is known as script level scoping. The scope of a variable can be changed with the use of the keywords “global,” “local,” “block,” and “script.” The scope of a variable or function is set when it is defined. For example, the following shows the creation of a globally scope variable:

var global myGlobalValue = 10;

Scope controls access to a program's data and to a lesser extent the functions that operate on that data. Program statements by default are scoped to script level. This means that any functions defined in a program script are only available in that program script and not in other scripts. Trying to call a function defined in another script will generate errors at compilation time. The exception to this rule is if the functions scope is set to Global. Globally scoped functions are available everywhere in the program. Setting a **functions** script level to Global does not apply to any variables defined in the function. Those variables will still be created at the script scope level unless a scope modifier is applied to the variable. Unlike OOPs languages the program has direct control over how their data is managed.

### Global

Global scope means that the variable is available anywhere in the program even across scripts. To access a global variable in another script you first specify the script where the global variable is defined followed by a period and then the name of the global variable. For example, if myGlobalValue is defined in a program script called setupModule001 then to access it and change its value you would type:

SetupModule001.myGlobalValue = 20;

### Script

Script scope is the default scope and is more restrictive than global scope. Script scope means the variable is available only in the program script in which it is defined. var variable001;

print (variable001);

Script scope variables can’t contain a period because they only exist within the script in which they are defined.

### Local

Local scope is only possible for use with variables and not functions. Local scope restricts the variables access to the the function in which it is defined. For example, in program003:

var MyFunction()

{

var local myLocalValue = 10;

If (a == 10)

{

print (a);

}

print(a);

}

The variable a can be accessed at several places within the function. However, outside of MyFunction(), the variable a cannot be accessed.

Local variables must not use the period access modifier as they only exist within the function in which they are defined.

# Identifiers

Identifiers represent named program entities. For example, in the statement var mine; the word “mine” is an identifier. Identifiers must begin with either an underscore or alpha character. After the first character, an identifier may contain any number of underscores, letters, digits, or UTF8 characters with a value > 127.

# Multiple declarations

Multiple variables can be declared one after another by separating them with commas. For example,

var variable1, variable2 = 10, variable3 = 25;

Can all be declared with a single var key world. When multiple variables are defined the names of the variable must follow the comma separating each variable definition. The last definition in a multiple variable definition must end with a semicolon.

# Classes and Modules

A module in the DSL is a single script file. This is similar to defining a class in other programming languages. The functions and variables that make up a script file are known as a module. By default all functions and variables defined within a module are scoped privately to the module in which they are defined. To make a variable, collection, or function visible to other modules it must be set to global.

# The DSL does not provide classes in the traditional programming language sense. However, the concept of organization provided by classes exists in the DSL. When multiple program scripts are compiled, the variables and functions within a program script can be available to others. This is accomplished by making them globally available. This ability is sometimes referred to as C with classes. For example if the function Circle exists in a file named Drawing to call that function you would use the statement, Drawing.Circle(5); The member access operator . tells the DSL that you want to access a global function in another module. *Note: Within the same file you can simply use the name of the function even it has been marked global.*Collections

## Strings and Search Expressions

The DSL does not provide regular expression syntax, instead if provides a simplified expression syntax for searching for groups of characters that match up closely with the C language character functions. This search expression syntax is easy to understand and use and can handle most cases where a search for a group of characters within a string is needed.

The DSL provides the following set of functions to simplify working with strings.

### string.find

The string.find() function searches the provided string beginning at the start location for the expression string. If the expression is found, its location is returned. If the expression is not found -1 is returned.

### var string.find(string, expression, start location)

|  |  |
| --- | --- |
| Search string | A string of characters. |
| Expression | A string of characters that may contain special expression characters that tell the DSL how to search the provided string. See the expression section. |
| Start location | A integer that indicates where in the string to start searching. |

### Returns

The zero based location of the first occurrence of the expression or -1 if the expression is not present in the string.

### Expression

The simplest expression string is simply a string of characters. For example,

print( string.find(“There are 3 foxes and 20 cats in the garden.”, “foxes”, 0));

would print the number 12, since the word foxes begins at location 12 in the string.

A more complex expression would be:

“%D”

This means look for a number from 0 to 9. This would match the 3, the 2 and the 0 in the string. For example:

print( string.find(“There are 3 foxes and 20 cats in the garden.”, “%D”, 0));

would print 10 since the number 3 exists at location 10. If the start location is changed to 11 then 22 would be returned which is the starting location of the word 20. For example:

print( string.find(“There are 3 foxes and 20 cats in the garden.”, “%D”, 11));

The % character is used in the expression string to indicate that the next character represents one of the character sets in the table shown here:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Description** | **Finds characters** | **C equivalent function** |
| %C | Control characters | 0x00 though 0x1F and 0x7F | Iscntrl() |
| %B | Blank characters | ‘\t’ and ‘ ‘ | Isblank() |
| %S | Space characters | ‘\t’, ‘\n’, ‘\f’, ‘\v’, ‘\r’ | Isspace() |
| %U | Upper case letter | ABCDEFGHIJKLMNOPQRSTUVWXYZ | Isupper() |
| %u | Lower case letter | abcdefghijklmnopqrstuvwxyz | Islower() |
| %A | Any letter | ABCDEFGHIJKLMNOPQRSTUVWXYZ  abcdefghijklmnopqrstuvwxyz | Isalpha() |
| %D | Any digit | 0123456789 | Isdigit() |
| %N | Any letter or digit | ABCDEFGHIJKLMNOPQRSTUVWXYZ  Abcdefghijklmnopqrstuvwxyz  0123456789 | Isalnum() |
| %P | Any punctuation | !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~ | Ispunct() |
| %G | Any character that can be seen when displayed. | !"#$%&'()\*+,-./  0123456789  :;<=>?@  ABCDEFGHIJKLMNOPQRSTUVWXYZ  [\]^\_`  Abcdefghijklmnopqrstuvwxyz  {|}~ | Isgraph() |
| %p | Any printable character | !"#$%&'()\*+,-./  0123456789  :;<=>?@  ABCDEFGHIJKLMNOPQRSTUVWXYZ  [\]^\_`  Abcdefghijklmnopqrstuvwxyz  {|}~ | Isprint() |
| %X | Any hexadecimal digit | 0123456789ABCDEFabcdef | Isxdigit() |
| %? | Any character in this place | While card matches any character |  |

To look for a % character add another % for example, %%. Multiple character sets can be searched for by adding multiples. For example,

print( string.find(“There are 3 foxes and 20 cats in the garden.”, “%D%S%u”, 0));

would return 10 but this line:

print( string.find(“There are 3 foxes and 20 cats in the garden.”, “%D%S%u”, 11));

would not return 20 cats because the 0 character does not match a space character.

### string.len

Gets the length of the string in characters.

### var string.len(string)

|  |  |
| --- | --- |
| Search string | A string of characters. |

### Returns

The number of characters in the string. 0 is returned if the string is empty.

### string.sub

Returns a sub string of characters.

### string.sub

Returns a sub string which begins at the start location and contains length characters.

### var string.find(string, expression, start location)

|  |  |
| --- | --- |
| string | A string of characters. |
| start | An integer indicating the location of the first character in the string to return. |
| length | An integer indicating the number of characters in the sub string to return. -1 means return all characters up to the end of the string. |

### Returns

A string of characters beginning at start and continuing for length characters. If length would extend past the end of the string only the characters up to the end of the string are returned. If the start location is after the end of the string, then an empty string of 0 length is returned. If the start location is before the start of the string, then an empty string of 0 length is returned.

### string.replace

Replaces all occurrences of expression in the search string with the replace string if the expression string is found.

### var string.replace(search string, expression, replace string, start location)

|  |  |
| --- | --- |
| Search string | A string of characters. |
| Expression | A search expression, See the string.find() functions expression parameter for details. |
| Replace string | String to replace if the search string is found. |

### Returns

A copy of the original string with any replacements made.

### string.tolower

Changes all uppercase characters in a string to lower case.

### var string.tolower(string)

|  |  |
| --- | --- |
| String | A string of characters. |

### Returns

A new string with all uppercase letters changed to lowercase.

### string.toupper

Changes all lowercase characters in a string to uppercase.

### var string.toupper(string)

|  |  |
| --- | --- |
| String | A string of characters. |

### Returns

A new string with all lowercase letters changed to uppercase.

### string.trimEnd

Removes characters from the end of the string if they match the provided expression.

### var string.trimpEnd(string, expression)

|  |  |
| --- | --- |
| String | A string of characters. |
| Expression | An expression with a single expression type that indicates what kind of characters to remove from the end of the string |

### Returns

A new string with any characters that match expression removed from the end of the string, if there are any at the end of the string that match expression.

### string.trimStart

Removes characters from the beginning of the string if they match the provided expression.

### var string.trimpStart(string, expression)

|  |  |
| --- | --- |
| String | A string of characters. |
| Expression | An expression with a single expression type that indicates what kind of characters to remove from the end of the string |

### Returns

A new string with any characters that match expression removed from the start of the string, if there are any at the beginning of the string that match expression.

### string.toCollection

Translates the string into a collection. The string must be formatted in JSON format.

### var string.toCollection(string)

|  |  |
| --- | --- |
| String | A string of characters in JSON format. |

### Returns

A collection if the string contains valid JSON. If the JSON format is invalid an error is returned.

### var string.fromCollection(collection)

|  |  |
| --- | --- |
| String | A string of characters in JSON format. |

### Returns

A string formatted as JSON representing the collection.

## Collections

Collections are a powerful data structure for organizing and managing the data. At its simplest level, a collection is a group of values managed as a single item. Collections must be created before they can be used. A collection is created by assigning the open and close braces to a variable when that variable is used. For example:

var a = {};

Adding or replacing values in a collection is as simple assigning something to it. For example,

a[“Ten”] = 10;

Would assign the number 10 to the collection referenced by the variable a. Unlike other variable types, a collection is always passed by reference. In this code the variable b references the a collection that was used in the previous example,

var b = a;

If the program changes b element “Ten” then the a collection element “Ten” will also be changed. For example,

b[“Ten”] += 10;

Would end up changing the a collection’s element “Ten” to 20 since the number 10 is added to the 10 that already exists in the a collection at element “Ten”.

Collections are only limited by the amount of resources on a system. This means they can grow as large as a program requires. The same scoping rules that are used for variables and functions also apply to collections. So once a collection goes out of scope, the resources it uses are returned and can be reused if needed.

By default, a collection can continue to grow to the limit of the resources provided by the computer system. In some cases, it may be desirable to limit how large a collection can grow. This is accomplished by calling the limit function. For example,

var v = { };

limit (v, 10);

Limits the collection to no more than 10 elements. Adding more elements would generate an error if an attempt to extend the collection past 10 elements is attempted.

The collection limit can be changed if the collection is in scope.

## Reading from a collection

The values stored in a collection can be read directly as in the previous example or in a specific order by adding an orderby() call. For example, if we have a collection containing the values:

var ac = {9, 1, 2, 3, 4, 5}

print(ac.orderby(ascending);

Would order the collection in ascending order so the numbers would be printed as:

{ 1, 2, 3, 4, 5, 9 }

*Note: That print automatically handles printing the collection. This also includes display of the keys associated with each value in the collection.*

## Removing Collection Elements

Removing elements from a collection is also very simple. To remove an element from a collection you set its value to null. For example,

ac[1] = null;

print(ac.orderby(ascending);

Would print the collection as:

{ 2, 3, 4, 5, 9 }

Since the element containing the number 1 has been removed.

Collections can contain other collections. This is similar to multi-dimensional arrays in other languages. In the DSL there are three ways of creating multi-dimensional collections:

1 – Create the collection inside loops,

2 – Use the create function to create a collections of collections for you.

3 – Use enclosing curly braces to create and initialize the collection.

For example, the following script could be used to create a 2 x 2 array to represent a simple math matrix:

var matrix = {};

For(var xx = 0; xx < 2; ++xx)

{

Matrix[xx] = { }

for(var yy = 0; yy < 2; ++yy)

{

Matrix[xx][yy] = 0.0;

}

}

Method 1 is similar to how other languages would create a multi-dimensional array. In the DSL a multi dimensional collection can be created by calling a collections create function. For example,

var matrix.create(2, 2);

The last method of creating a collection also allows the collection to be initiated with values. For example,

var matrix = { { 1.0, 1.1 }, { 2.0, 2.1} };

### Working with Collections

Collections can be accessed like arrays in other languages, but the elements of a collection are not stored as array elements. Each element in a collection is stored in a hash table. This means that it is not necessary to loop over a collection to find an element in the collection. If you know the key for an item in the collection you can retireve that value very quickly. This also means that a collection element does not need to be placed at a numerical index. For example,

var myCollection = {};

myCollection[“MyAwards”] = { “a1”, “b2”, “c3” };

myCollection[“OtherAwards”] = { “d1”, “e2”, “f3” };

Would create a collection where the awards are keyed by the strings “MyAwards” and “OtherAwards”. For example,

print(myCollection[“MyAwards”]);

Would print all of my awards on the console.

If you do not specify a key when assigning a value to a collection, the DSL creates one for you. This key value is calculated from the number of elements in the collection. For example,

var myData = {};

myData[] = 10;

myData[] = 20;

Would add the number 10 to the end of the collection. When the collection is created it automatically has a single element with a key of 0. So the value 10 would be added with a key value of 1. Likewise the value 20 would be added with a key value of 2.

The key assigned to a collection can be replaced by assigning a new key value to an existing key. For example, using the collection from the previous example,

myData[0=3] = 3;

Would change the first assigned values key from the automatically assigned 0 to 3.

The keys a collection uses can consist of multiple collections. In relational databases these are known as composite keys.

var c1 = { “word1”: “hello” };

var c2 = { “word2”: “World” };

var result = { “word1.word2”: c1 + c2 };

### Collection Group Operations

Collections allow you to perform groups of operations on a list of values. For example,

var a = { 100.00, 125.00, 300.00 };

a \*= .75;

print(a);

would multiply each element of the values in the collection referenced by variable a by the number .75 and assign the result to the a collection.

Operations on collections can also be performed between collections. For example,

var a = { 200, 300, 400 };

var b = { 1000, 1100, 1200 };

var c = a + b;

print(c);

Would add each element in the a collection with each element in the b collection and assign the result to the c collection. So the previous example would print { 1200, 1400, 1600 }.

If the collections in the expression have different sides a minimum is calculated and only those elements within that minimum amount are updated.

### Typeof

Type of is used with values to indicate the kind of information. This can be integer, double, character, string, or bool.

### Adding Replacing, and Deleting elements

Anytime you assign a value to a collection and the key for that element does not exist a new element is added to the collection. If the key does exist the element in the collection is updated. If you assign the value nul to an element in a collection, the collections element is removed. For example, to add a value to the myData collection,

myData[4] = 4;

Would add a new element to the myData collection. The following would replace the value with the key 1 in the myData collection,

myData[1] = 1;

Removing an element from the collection can also be accomplished as shown here,

myData[2] = nul;

You can also replace a value in a collection and change that items keys by adding the key replace key while assigning a value. For example,

myData[3=5] = 5;

Key assignments and collection element access to no have to be scalar values. You can use variable values as shown here,

var newCollection = { };

var stuff = “MyStuff”;

newCollection[stuff] = { “Wrench”, “Hammer”, “Nail” };

stuff = “YourStuff”;

newCollection[stuff] = { “Computer”, “Mouse”, “Desk” };

### Value Retrieval

Retrieving a single element in a collection is straightforward and uses syntax like accessing elements of an array. You can also retrieve multiple elements from a collection. For example,

var states = {

“EST”:{

{ “VA”, { “Mclean”, “Richmond” } },

{“MA”, {“Ocean City”, “Rockville” } },

},

“PST”:{

{ “WA”, {“Seattle”, “Redmond” } },

{ “CA”, { “Losangles”, “Sanfranciso” } },

};

print(states[“EST”]);

Would display only the states and cities in the eastern time zone in the order they were added to the collection. You can override the order in which collection information is retrieved by using the orderby function. For example, using the previous collection the retrieved information can be displayed in ascending alphabetial order with the following line of code,

print(states[“EST”].order(0, true));

The order(0) tells the DSL that you want to sort the retrieved data on the first element of the retrieved data. In this case that is the state prefix. The second parameter to the order function specifies you want the data retrieved in ascending order.

print( states[0] );

Would display Mclean Richmond

You can also create an association to some data in the collection by assigning instead of calling print. For example,

var westernStates = states[1];

Order

… orders a retreive set of collection values in ascending or descending manner

Where(collection elements, expression)

… restricts a collections retrieved values where a set of keys matches the expression

### Composite Keys

Multiple keys can be combied into a composite key. A composite key contains the parts of the key separated by commas. For example, using the previous collection the cities that are in the eastern time zone for the Maryland state could be printed as shown here:

print(states[“EST”, “MA”]);

When EST and MA are passed to the DSL, the DSL constructs a composite key and uses it to retreive the information which is then printed.

## Saving a collection

Call write to save a collection or any set of retrieved values from a collection. The data will be saved in JSON format.

## Reading a collection

Call write to read a collection. The read method will translate the JSON format finput into a collection and return those values to your program.

## Write

Collections can be written to a by calling the Write function. For example,

var c = { 1, 2, 3 };

Write(“https://www.someserver.com”, c);

Each value in the collection will be written to the server at someserver.com in JSON format. For the previous example this would result in a JSON file containing:

{

"0": "1",

"1": "2",

"2": "3"

}

## Read

Collections can be read from another resource by calling the Read function. The file is read and the information in the file is returned as a JSON formatted collection.

## Print

Collections and elements of a collection can be printed to the standard output device by calling the print function.

## Input

Information can be entered in the standard input device and this information is returned as a collection.

Read, Write, Print, Input

Collections can be Read from a file, written to a file, Printed, even entered via input. When a collection is written to a file it is written in serialized JSON format. This allows for easy data interchange with other applications and systems. Reading is similar and the returned collection will represent the JSON format of the file.

# Operators

The DSL is an operator rich language allowing expressions to be succinctly specified. The order processing of operators determines how the expressions in the DSL are interpreted by the language compiler. Anyone familiar with modern programming languages will be immediately familiar with the precedence of the operators that the DSL provides. The table below outlines the provided operators and their order of precedence.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbol | Type | Precedence or  Binding power | associativity | Remarks |
| {} | Block | 16 | Left to right | Defines a group of statements or items in a collection. |
| () | grouping | 15 | Left to right | Groups terms in an expression, or parameters in a function. |
| [] | access | 15 | Left to right | Access element in a collection. |
| \*\* | Exponetiation | 14 | Right to left | Raises the left expression to the power of the right expression. |
| ++ | prefix | 13 | Right to left | Increment the next variable by 1. |
| -- | prefix | 13 | Right to left | Decrement next variable by 1. |
| + | Unary | 13 | Right to left | Next value is positive. |
| - | Unary | 13 | Right to left | Next value is negative. |
| ! | Unary | 13 | Right to left | Logical not toggles true/false state. |
| (int) | Unary | 13 | Right to left | Converts the next value to an integer. |
| (double) | Unary | 13 | Right to left | Converts the next value to a double. |
| (char) | Unary | 13 | Right to left | Converts the next value to a UTF8 character. |
| (string) | Unary | 13 | Right to left | Converts the next value to a string. |
| (bool) | Unary | 13 | Right to left | Converts the next value to a boolean. |
| \* | Multiply | 12 | Left to right | Multiply the left and right expressions. |
| / | Divide | 12 | Left to right | Divide the left and right expressions. |
| % | Modulo | 12 | Left to right | Take the remainder of the left and right expressions. Note: floating point values will generate an error. |
| + | Addition | 11 | Left to right | Add the left and right expressions. |
| - | Subtraction | 11 | Left to right | Subtract the right expression from the left. |
| << | Bitwise Shift Left | 10 | Left to right | Shifts the bits in the left expression to the left. The number of places shifted are specified in the right expression. |
| >> | Bitwize shift right | 10 | Left to right | Shifts the bits in the left expression to the right. The number of places shifted are specified in the right expression. |
| < | Less than | 9 | Left to right | True if the left expression is less than the right. |
| <= | Less than or equal to | 9 | Left to right | True if the left expression is less than or equal to the right. |
| > | Greater than | 9 | Left to right | True if the left expression is greater than the right expression. |
| >= | Greater than or equal to | 9 | Left to right | True if the left expression is greater than or equal to the right expression. |
| == | Equal to | 8 | Left to right | True if the left expression and the right expressions are the same. |
| != | Not equal to | 8 | Left to right | True if the left expression and the right expressions are not the same value. |
| & | Bitwise AND | 7 | Left to right | Performs a bitwise AND operation with the bits in the integer expression. Any bits set to 1 in both values remain 1 all other bits are set to 0. |
| ^ | Bitwise Exclusive OR | 6 | Left to right | Performs a bitwise exclusive OR operation with the bits in the integer expression. Any bits that are different are set to 1, any bits that are the same are set to 0. |
| | | Bitwise Inclusive OR | 5 | Left to right | Performs a bitwise OR operation on the bits in the integer expression. Any bits that are 1 in either the value remain on all other bits are set to 0. |
| = | Assignment | 4 | Right to left | Copies the value or result of the expression on the right side to the variable on the left side. |
| -> | Reference | 4 | Right to left | Creates a pointer or reference to the variable or collection on the right side. Any changes to the left side variable also effect the right side variable. |
| \*= | Assignment | 4 | Right to left | Assigns the product of the right expression and the left variable to the left variable. |
| /= | Assignment | 4 | Right to left | Assigns the quotient of the right expression and the left variable to the left variable. |
| %= | Assignment | 4 | Right to left | Assigns the remainder of the right expression and the left variable to the left variable. |
| += | Assignment | 4 | Right to left | Assigns the sum of the left variable and the right expression to the left variable. |
| -= | Assignment | 4 | Right to left | Assigns the difference of the right expression from the left variable to the left variable. |
| ++ | postfix | 3 | Left to right | Increment the previous variable by 1. *Note: Postfix operations are applied after all other expressions have been evaluated.* |
| -- | postfix | 3 | Left to right | Decrement the previous variable by 1. *Note: Postfix operations are applied after all other expressions have been evaluated.* |
| , | comma | 2 | Left to right | Expression terminator forces the expression to be evaluated. |
| ; | Statement terminator | 1 | Left to right | Statement terminator forces immediate evaluation. |

# Expressions

Expressions are formed by combining values, variables, functions, and operators. The end of an expression is indicated by a semi-colon, comma, and in some cases a close parenthesis. For example, the DSL script shown here contains several expressions outlined in bold:

while (**f < 100**)

{

**F += 5;**

**c = (70-32) \* 5/9**;

print (**c**)

}

# Order of Evaluation

The DSL parser uses precedence climbing which evaluates expressions based on their evaluation order and precedence or binding power. This approach is known as Pratt Parser named after its inventor Professor Vaughan Pratt. The print of the expression evaluation is added to the AST which is rasterized as a flowchart within the IDE designer.

## Postfix Increment, Decrement

In C the evaluation of post increment and decrement operators is left up to the compiler implementor. Over the years the way post increment operations work in C has been unofficially standardized so that post increment and decrement operators are applied after the other operations The benefit of this approach is that it allows expressions to be evaluated and then the indexers adjusted to prepare for the next evaluation. In the DSL **post increment and decrement operators are evaluated after the other operations have been evaluated**. For example, in the expression a = b++ + 5, that b + 5 is evaluated and the result assigned to a before 1 is added to b. Postfix decrement operations work the same way.

## Unary operators

The DSL provides three unary or single value operations. These operations are shown here:

|  |  |
| --- | --- |
| Symbol | Description |
| + | Unary Positive, Multiplies the following target by 1. Note: In the compiler unary positive is ignored since 1 \* any value results in any value. |
| - | Unary minus, multiplies the following expression by –1. |
| ! | Unary NOT, inverts the following value. If the value is true, it will become false. If the value is false, it will become true. *Note: Values other than true and false will be converted into Boolean true and false values before the Unary NOT is applied.* |

All unary operations must be followed by a valid target. This target can be a variable, value, or function call. For example, all the following are valid unary operations:

a = !GetResult(1, 2);

b = -55;

c = +100;

d = (1 + -1);

In the DSL unary operations must have some immediate value to work with. This means that unary operators cannot be strung together into chains of unary operations. For example,

a = !!5;

Generates an error since either the first or second not operation does not have anything to work with. Another example is shown here:

a ---5;

In this example, the unary minus and unary minus is a pre decrement operation and will generate an error since pre decrement requires a variable to assign the result of the operation.

Unary operators are evaluated from the right side of the expression to the left side of the expression in the order they occur. For example, in the following expression:

b = 5;

a = 5 - -b; //notice the space between the – and –5.

-b is evaluated first. This changes the value in b to 5, resulting in:

a = 5 - -5;

Adding 5 and –5 results in the value 10 which is assigned to variable a. Another simplified way of looking at this expression is:

a = 5 - (-b);

or

a = 5 + 5;

Notice in each case the –b is evaluated before the binary minus expression is evaluated. Another example,

a = 10;

a = -5 + -6;

In this case the 6 is turned negative and then the 5 is turned negative. This is because the expression is evaluated from right to left for unary operations. Once all of the unary operations have been calculated, the results are added together resulting in –11 being assigned to the variable a.

## Modifiers

### const

Declares that the variable is immutable (read only) and cannot be changed. For example:

var distanceToField const = 100.0;

*Note: Because all variables are automatically assigned a default value when defined, it is not an error in the DSL to apply the const modifier to a variable without an expression.*

### global

Declares that the variable or function is accessible everywhere even outside the current program.

### script

Declares that the variable or function is accessible everywhere within the current script. This is the default scoping rule and is applied if the scope of the variable is not specified.

### local

Declares that the variable is only accessible within the current function.

### block

Declares that the variable is only accessible within the current statement block.

## Declaration

### var

Declares that the next identifier is a variable or the return type of a function, that can contain any type of known data. An assignment expression is required after the identifier to set the variables type. An error is generated if the assignment expression is not present.

## Branching

Branch statements enable the program to make decisions. The following decision statements are supported:

### If

The if statement evaluates the following expression and if the expression evaluates to true the statement block following the if statement is executed. If the expression is evaluated to be false, the following statement block is bypassed and the following statement block is executed. The expression must be enclosed within parentheses an error is generated if the parentheses are omitted. The expression must evaluate to a Boolean expression otherwise an error is generated.

### else

The else statement is optional and if used must follow the if statement block. If the if statement is evaluated to be false, then the else statement is executed. This causes the statement block following the else statement to be run. If the if statement evaluates to true, the statement block following the if is executed and the statement block attached to the else is bypassed. If and else statements can be combined to form complex if else statements.

### switch

The switch statement is used with the case and default statements. Like the if statement the switch statement evaluates the expression inside the parenthesis. The result then matched against the case values present and if one matches the statement block attached to that case statement is executed.

### case

The case statement must be used with the switch statement specifies a value to be matched against the expression in the switch statement, and a statement block to be executed if a match occurs.

### default

The default statement is similar to case except an expression is not provided. Default is optional and if included in the switch statement means run this block of statements if none of the other case statements match.

## Looping

Looping statements allow a block of statements to be repeatedly executed. The following looping statements are supported:

### while

The while statement evaluates the expression following the while keyword and if the expression evaluates to true the statement block attached to the while is executed. When the expression evaluates to false the statements after the while statement block are executed.

### for

For statement is a compound statement. It has three parts: initializer, conditional, update. Any part of the for statement can be omitted if not needed. Each part of the for statement is separated by a semicolon. The semicolons must be present even if the specific part is not present. The initializer expression is executed as soon as the for-loop statement is encountered. Next the conditional expression is evaluated. If the expression evaluates to true, the statement block following the for statement is executed. When the last statement in the statement block is encountered the update expression of the for statement is executed. This is followed by evaluation of the conditional statement, and if the result is still true, the statement block is executed. If the conditional statement evaluates to be false, the loop is exited and the statement following the for-statement block is executed.

## Control Flow

The following statements are used to change the default flow of statements within a statement block.

### break

Immediately exits the current statement block running the statement following the current block. An error is generated if the break statement is used outside of a statement block.

### Continue

Immediately jumps to the top of the current statement block and executes the first statement encountered. If the statement block is a for statement, then the update expression is executed.

### return

Evaluates the expression following the return and immediately returns it value to the caller. The return statement must be used instead of a function. Using return outside a function will generate an error.

# Functions

Functions define subroutines that can be called from various places in a program to perform some action. A function is defined like a variable with the keyword var . All functions in the DSL take a variable number of inputs. This means that a function can be called from one place and passed 2 inputs and in other places passed 1 or 3 inputs. An example of why this is useful is the built-in print function. For example,

print(“Hello”, “ World”);

print(“Hello World”);

Both calls to the print function display the text Hello World on the console, but they are called differently. A Function is a set of code statements that are not run until the function is called. To declare a function the keyword var is used. This is followed by the name of the function which must be followed by an open parenthesis. Like variables a scope modifier can be specified between the var keyword and the name of the function. Unlike variables functions can only be set to script or global scope level. Like variables, if no scope modifieris specified the function is set to local scope.

All functions in the DSL can take any number of arguments. Arguments are set to local scope which means they are only accessible from within the functions code block. For example, this function returns the sum of the first 2 values it receives:

var MyFunction() =

{

return $1 + $2;

}

A function can optionally specify the names of the parameters it expects to receive. For example:

var MyFunction(a, b) =

{

return a + b;

}

Allows the function to use a and b instead of $1 and $2. Internally there is no difference between $1, $2 and the specified arguments a, and b.

Functions once defined can be called from anywhere in the program. Unlike other programming languages there is no need to declare a function as existing before calling it.

# Warning Levels

Unlike most languages the DSL defines the mean of specified warning and error levels. This prevents confusion when different compilers, interpreter, and other tools are produced.

|  |  |
| --- | --- |
| Warning Level 0 | Warning messages are turned off. |
| Warning Level 1 | Displays warning messages that are unlikely to cause issues. Variables and functions can be declared without the var keyword. |
| Warning Level 2 | Displays level 1 warnings plus warnings that may cause problems. Variables and functions declared without the var keyword will generate a warning. |
| Warning Level 3 | Displays all level 2 warnings plus severe warnings that are likely to cause problems. |
| Warning Level 4 | Warnings are treated as errors. |

# Standard functions

Because most programming languages were developed before standardization of information interchange became commonplace, other programming languages avoid defining methods for information transfer leaving these details up to the language consumer. This has provided a wealth of custom ways to move information from application to server and back making the programming environment more complex and error prone. The DSL takes a different approach to this problem by specifying how information is moved between applications and servers.

abs

acos

asin

atan

atan2

cos

sin

tan

cosh

sinh

tanh

Exp

log

Log10

sqrt

ceil

Fabs

floor

Fmod

## random(low, high)

Returns a random number between low and high values inclusive.

## seed(low)

Sets the seed used by the random number generator.

## tick()

Gets a value indicating the time since the operating system was last started. This value can be passed to the seed function to seed the random number generator.

## print (expression, expression, expression, ...)

The print function sends the expressions to the local device. By default, this is the display standard output terminal. Note: Eventually this will include graphical assets and allow 2D, 3D and other asset types to be ‘printed’. Note: printf can be used as alias for the print function. *Note: printf() can also be used, the DSL treats print and printf the same way.*

## input()

The input() function reads text input from the standard input stream. This is normally the console and is provided by the DSL for debugging. To read input from a device the event system should normally be used.

## read(from)

Reads text and JSON formatted data from the specified location and returns it. The information is returned

## write (destination, expression, expression, expression)

The write function contents of the expressions to a destination. This can be a local file, or a remote server setup to receive information. The file is written in JSON format.

## files (location)

Returns a collection listing the files at the specified location. The format is URL and can specify a remote and local file location. The file is returned as a collection.

## delete(location)

Deletes the named resources at the specified location.

# Events

Events are messages or signales that are sent to your program when specific actions occur. For example, pressing or releasing a key on the keyboard. When an event occurs the runtime checks to see if your program contains an event handler. All event handlers begin with the word On followed by the name of the event. For example, to be notified when something happens with the mouse you would add the following event handler to your program:

var OnMouse(left, right, x, y)

{

//... do something

}

Events are scope level functions. This means that each script in your program can have its own event handler. When the event occurs it is sent to each handler function. You have the option in your event handler to let the runtime know if the event should continue to be sent to other handlers, or not, by setting the function events.Stop(“event name”) in your event handler function.

## Calling Order

By default event functions are called in the order the program scripts are compiled. This order is the same as the order they are specified in the build.json file for your project. If your project does not have a build.json file the compiler will build one for you the first time that the compiler is run. By modifying the build order in the json file you can force the order that the run time will call each event.

## User Events

The DSL event system also lets you create your own user defined events. To create an event in your program you simply create a function that begins with the word **On.** Case is significant which means the function must start with On exactly as written. This is then followed by the name you want your event to be called. For example, to create a user event handler in your program you could write:

var OnMySpecialEvent()

{

}

If you try to call this event function directly the compiler will generate an error. Event functions can’t be called directly. Instead you have to call the event by calling it though the events program. For example, to call the event created in the previous example, you would add code like this:

Events.MySpecialEvent(…);

## System Events

The DSL rumtime has system defined events that you can use in your programs.

### OnError

Called when an error occurs.

### OnMouse

Called when the local system mouse is clicked, moved, or performs some kind of drag operation.

### OnKey

Called when a key is held down, pressed, or released on the local device

### OnTick

Called at the rate of 1/20th of a second.

### OnInformation

Called when information is read or written. This information can be from the local device or from a remote location. The location the information where the information is originated is included in the parameter data sent to the event.

# Error handling

The DSL provides automatic error handling by outputing descriptive information about the issue to the console. A program script can enhance and customize this error handling behavior by adding a script level error handler. The error handler needs to return a value which tells the program script how to proceed. To add an error handler to a program script create a function and return one of the predefined defined error constants. For example,

OnError(code)

{

switch(code)

{

default: { return error.quit; }

case 100: { return error.continue; }

case 200: { return error.restart; }

case 300: { return error.reload; }

case 400: { return error.waitQuit; }

case 500: { return error.WaitReload; }

}

}

## Raising errors

The DSL will automatically handle many system and general errors, in some cases though you may want specific error handling to occur. In these cases your program should should not call OnError() directly. Instead it should Raise the error event by calling events.raise(“OnError”, true, …); The first parameter is the name of the event you want to raise. The second parameter is true if you want to notify all of the error handles in each program script or only the OnError handler in the script file that raised the error.Several global system variables can be set to customize the error handling process.

error.retryWaitTime Sets the amount of time to wait before retrying the code that caused the error to occur. Default time is 1 second.

error.retryCount Sets the maximum number of times error.retry can be attempted before the value becomes error.retryQuit Default count is 3 times.

error.continue returning error.continue will allow the program script to continue processing on the next line after the line where the error occurred.

error.quit

Returning error.quit will cause the running program to stop running.

error.restart

Returning error.restart will cause the current script to be cleared, reloaded, and start running from the beginning of the script.

error.retry

Returning error.retry will retry the code that caused the error.

error.waitQuit

Returning error.waitQuit will wait for the amount of time specified in the global variable error.waitTime before retrying the line of code that caused the error. The wait return can be used a maximum of error.retryCount times before the quitting. The default is 0 or no retries.

Error.waitReload

Returning error.waitReload will wait for the time specified in the error.waitTime variable before reloading the current program script and trying to run it again.

# Compiler Options

# Error Messages / Troubleshooting