OLCUS.NET 3.0

Reference Manual

FBI Science GmbH

For Better Innovation!

Version 3

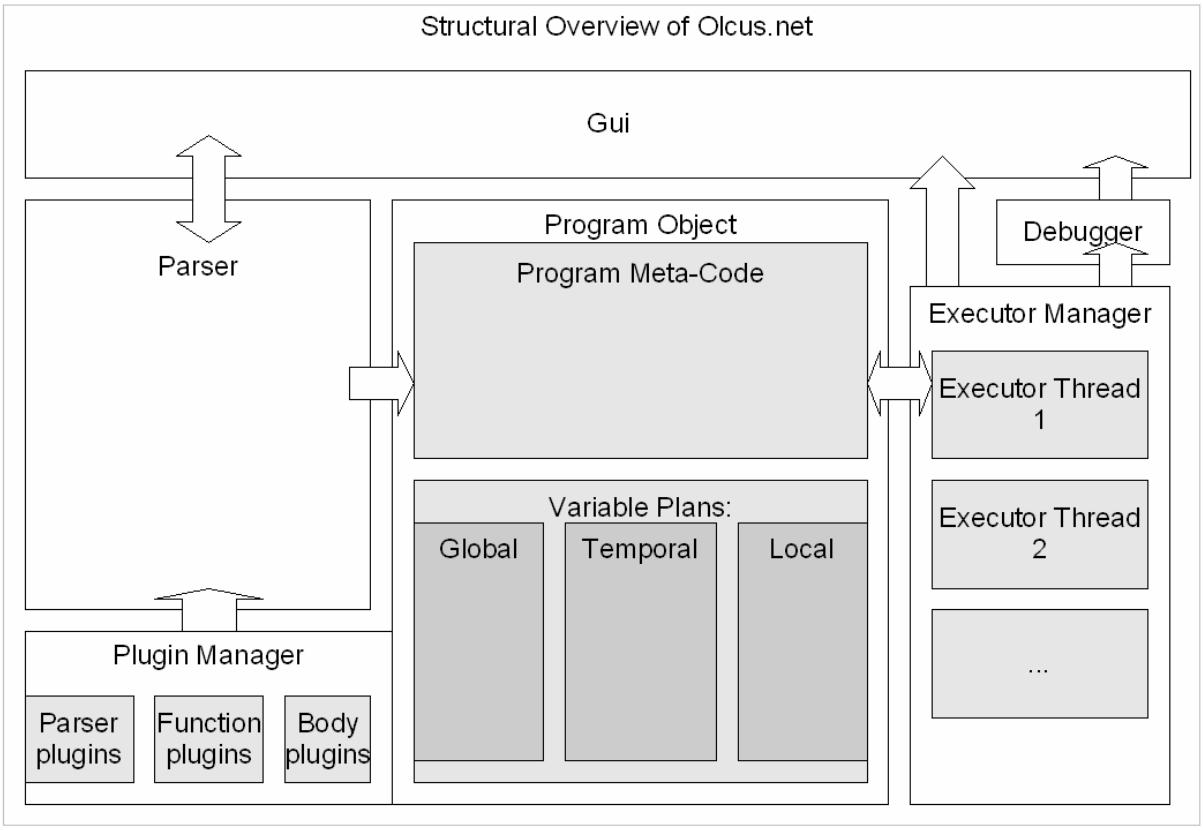
1 Table of contents

2 Aims

Olcus.net or OLCUS.NET 3.0 was designed with the following key aspects:

- Clean Multithreading: Several Olcus.net programs can be run simultaneously in the same session. All key objects can be instanced several times per session, with no side effects.
- Flexibility: Olcus.Net's clean object oriented structure allows the objects to be viewed from different perspectives; implemented parts can be replaced as needed without need to touch any of the core components.
- Extendibility: OLCUS.NET is designed with several Plug-in interfaces that allow the system to be extended to fit needs not covered by the basic functionality.
- 3rd party support. OLCUS.NET can be extended by 3rd party developers, using the Olcus.net SDK. Support for OLCUS sessions running these plug-in will not be granted unless plug-in are evaluated and signed.

3 Structural Overview



This section is supposed to give a brief overview of olcus.net's structure. Understanding the structure of olcus' internal workflow should be helpful to both, programmers of olcus program code, and plug-in developers.

3.1 Parser

The parser can accept Olcus program code and translate it into microcode that can be interpreted by an olcus executor. The parser can accept any character-stream, implemented at design time, however is only the file-stream. Files can include code from other files via the include command.

A compile process needs to complete without errors, so a program object becomes executable.

3.2 Program

The program Object is generated by the Parser. The program object consists of two main sections: The program meta code, which is executed at runtime, and the Variable plans.

3.2.1 Meta Code

Olcus meta coda consists of an array list of Command Objects. A Command object needs to be inherited from the Type core.commands.Command. For readability the Type Name should be starting with „Cmd“ so command objects can easily be picked.

While the Meta-Code Array can be accessed directly inside of the Program object, it should only be manipulated via the methods provided for that. Further details will be included in the plug-in development section of this document.

3.2.2 Variable Plans

Variable plans need to be used because at parse time the Variables do not yet exist, but the command objects need to know which variables they need to access. There for an abstraction layer implemented by the oPlan object was implemented. An oplan object knows where to find the variable it represents at runtime, so the process of variable access should still be more or less easy and transparent. The program contains all variable oPlan objects, in three Array Lists for three different variable Types:

3.2.2.1 Global

Global Variables are instanced once per Program. Global variables can only be defined in a „global“ section, of the program code. All running Executors share access to the same global variable. Therefore common multithreading problems may occur. (Concurrent access of the same variable)

3.2.2.2 Constants

Constants are automatically created by the Parser whenever constant values are encountered. (i.E. I = 12 will create a constant value „12“ so the assignment can be done by the microcode in the specific line.) Since constants cannot change, they are instanced only once per program, no multithreading problems are expected.

3.2.2.3 Temp

Temporal variables are automatically created by the parser to allow Microcode commands to pass values. Temporal variables will store in between results of complex operations, or will be used to pass a value from one function to another in a complex instruction line. Since Temporal variables need to be thread save they are instanced once per Executor, so interference between concurrently running Executors is not possible.

3.2.2.4 Local

Local variables are instanced once per executor and subroutine. Local variables are an exception in a lot of ways. Their plans are not really stored inside of the program object itself, but internally inside of a goSub MetaCommand. Local Variables are only valid within one subroutine, and loose their validity whenever either an end command or a sub command is encountered.

Example:

```
body
new(a integer) = 12
gosub(test)
end

sub(test)
console.echo(a) # invalid
end

body
new(a:integer) = 12;
gosub(test,a)
end

sub(test,b integer)
console.echo(b)   #valid
end
```

3.2.2.5 Internal

Internal variables are not shown to the actual program code, and are internally used by plug-in to store internal thread-context-sensitive information. The mechanism to store and retrieve internal variables will be discussed in the plug-in development section.

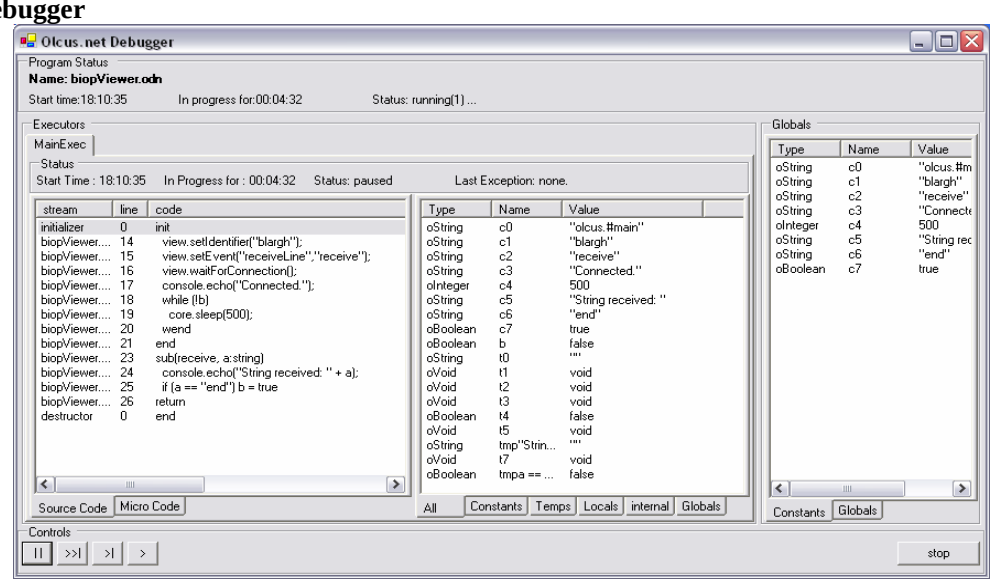
3.2.3 Executor Manager

The executor manager manages the execution of a program object. It will instance Executors as needed during the program execution. The debugger communicates with the manager to display debug information during a debugging session.

3.2.3.1 Executors

The executor represents a single thread of Program execution. The executor object contains an array list containing all local variables used to for program execution. On instancing the oPlan objects for temporal and local variables will be used to create according local variable stacks.

3.2.3.2 Debugger



The debugger displays information about currently running Sessions. The debugger displays information about all available Variable stacks.

The tap-panel to the left contains one tap per currently running executor.

Each tab contains information about the executors running time, the running code, and the current variables.

3.2.3.2.1 Code

The left part of the panel holds information about the currently executed code. The highlighted lines of code indicate the commands currently evaluated by the executor.

3.2.3.2.2 Source Code

The source code file the code originates from. The column „Stream“ indicates what Stream the current section of code originates from. „Line“ indicates the line the code is stored under. (includes might change the order of line numbers inside the code) and finally the column „code“ contains the character stream the parser used to create the microcode from.

3.2.3.2.3 Micro Code

This tab shows the microcode that was created from the code shown under „Source Code“. The column „pos“ indicates the position of the meta command inside the command array inside of the program object. „Name“ indicates the generic name of the meta command. „operators“ gives an overview of the operators this command uses. Finally „Fragment“ contains the fragment of source code the meta command was created of.

3.2.3.3 Variables

The right panel shows an overview of the current Variable stack.

3.2.3.3.1 All

This tabs shows an overview of all variables valid for the current executor.

3.2.3.3.2 Constants

This tabs shows all current constants active for the Program. A copy of this Tap is available in the panel right to the executors panel.

3.2.3.3.3 Temps

This tabs shows all temporarily variables currently valid for the executor. Temp Variables are instanced per Executor, so no inter thread operation problems occur.

3.2.3.3.4 Locals

This tab shows all variables currently valid for the subroutine currently running inside of the executor.

3.2.3.3.5 Internal

This tab shows a string representation of all currently internally stored objects. (Usually stored for internal communication by plug-ins) The String representation might give a vague idea of the content of an object at best. These objects are better examined while debugging the actual plug-in, inside of a .net ide.

3.2.3.3.6 Globals

This tab shows all global variables valid for execution of the current program. A copy of this tap exists in the panel to the right.

3.2.3.4 Controls

The program execution can be controlled by the panel below the debugger display.

3.2.3.4.1 '||' Pause

Pauses execution of the program. All executors will suspend evaluation of commands, until instructed otherwise.

3.2.3.4.2 '>>|' Single Line Step

Executes one line of code from the original source code. That might be several Meta-Commands at once.

3.2.3.4.3 '>|' Single meta Command

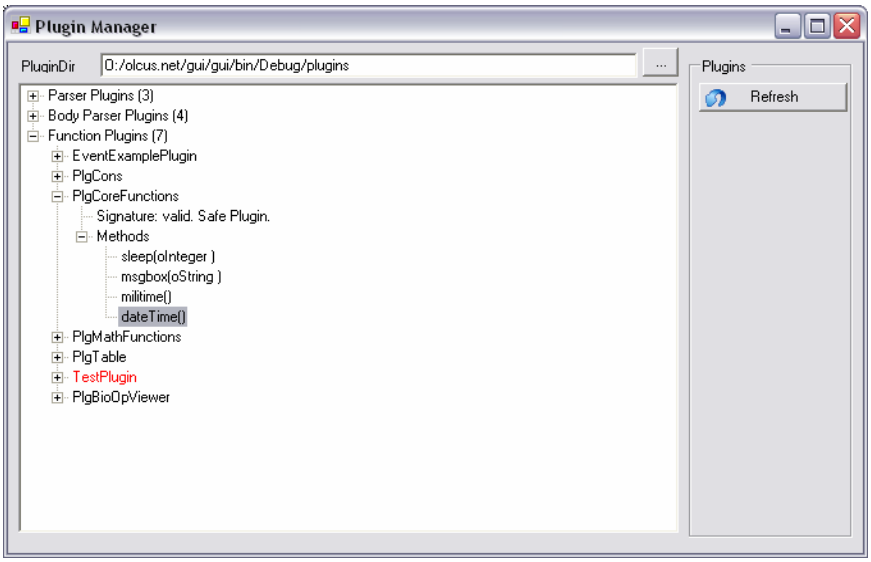
Executes only one Meta-Command at a time, then pauses execution again.

3.2.3.4.4 '>' Play

Executes the rest of the command until either the Program ends, or pause is pressed.

3.2.4 Plug-in Manager

The plug-in manager can be used to gain an overview of all plug-ins currently available to the olcus.net application and their status.



3.2.4.1 Function Plug-ins

The node function plug-ins contains information about all plug-ins currently extending olcus' function calls. Each plug-in node contains information about available methods, their parameters, and plug-in events. This can be useful during code creation, for easy access to keywords necessary for making successful function calls.

3.2.4.2 Section Plug-ins

This node contains information about all currently installed section plug-ins.

3.2.4.3 Body Section Plug-ins

This node contains information about all installed body section plug-ins and their status.

3.2.5 Services

The maintenance object offers several useful services.

3.2.5.1 Logger Service

The logger service is a plug-in based central logging service.

3.2.5.1.1 Out putters

Logging messages can be routed into various destinations, using Out Putter classes. Several Out Putter classes may be stacked to direct the Stream into various destinations at once. Each out putter plug-in can decide on its own if the messages priority is relevant to it.

3.2.5.1.1.1 LogOutToFile

This directs the Log stream into a file. New messages are automatically appended. If the file exceeds a certain size, a zip-compressed archive will be created and the log file will be recreated.

3.2.5.1.1.2 LogOutToConsole

This outputs the messages to the Standard output. (console)

3.2.5.1.1.3 LogOutToListBox

This will add messages to a List box provided to the constructor of the Out putter.

3.2.5.1.1.4 LogOutToTextBox

This will add messages to a TextBox provided to the constructor of the Out Putter. In contrary to the list box out putter one message my occupy several lines.

3.2.5.1.2 Priorities

Messages to the logging service are posted with .log(priority, message). The following priorities are available as static integers of the Logger object. The method .setLogLevel(int) allows to set a filter for logging messages. All messages below the selected log level will then be ignored by the logger. (unless an out putter implements its own priority check). The following log levels are available, ordered by their priority level:

3.2.5.1.2.1 Logger.DEBUG

Messages used during the debugging state of developing plug-ins.

3.2.5.1.2.2 Logger.INFO

Non Vital Information about the program execution

3.2.5.1.2.3 Logger.WARNING

Warnings about states that might cause problems, but are not considered errors.

3.2.5.1.2.4 Logger.ERROR

Errors that occur. Usually at least this log level should be set.

3.2.5.1.2.5 Logger.CRITICAL

Critical messages, of events that might hinder correct execution of the program, or that are otherwise important.

3.2.5.2 Configuration Service

The configuration service provides globally available configuration values. The Service is auto-adaptable. Once a value is queried that is not available in the configuration file it is automatically added.

The default value for each configuration value is literary „default“. If a value is set to default, the default value will be used that is hard coded into the program. If default is overwritten by a different value, that value will be used instead.

The configuration file can be edited with the configuration service available from the olcus main form. Or by instancing the type „support.ConfigDialog“ and providing the configuration object to the constructor.

4 Script Development

This section will give an overview about how to write a script controlling your data. You will find many similarities to other programming languages, which should make it easy to adapt to.

4.1 Olcus Variable Types

Before the details of plug-in development will be discussed, a quick overview of olcus variable system is necessary.

All Olcus variables are inherited from „oVariable“. Per definition every variable has a „.get()“ method, that will extract the base type of the variable, so the plug-in function can work with the context. The get function returns the natural base type, no override will be necessary.

The counterpart is the .put() function that can be used to set the variables internal base type.

Creating an Olcus Variable is as simple as using the new command with the default constructor. `new oString(„hello world“)` will create a new String object that can be safely returned to the olcus engine. `new oString()` will create an empty string object, that can be filled with a value using the .put method.

For methods that do not return anything, the oVoid variable needs to be used. For convenience the oVoid Type has a static method called getVoid() which returns a freshly instanced oVoid variable that can be returned to the olcus engine.

4.2 Code Structure overview

Olcus code is structured in Sections. Sections might contain very different styles of instructions. This document will focus heavily on the „body“ section, which usually contains most of the instruction code of an olcus.net code.

4.2.1 Sections

Available sections are:

4.2.1.1 Global

This section allows the definition of global variables. The syntax is the following:

<type> <alias>[= <initialization value>]

4.2.1.1.1 type

The variable type. Though extendable, those variable types are at least available: Integer, Float, String, Boolean. These work analog to these variable types in other programming languages.

4.2.1.1.2 alias

the Alias you want the variable to be known under inside of the body section.

4.2.1.1.3 Initialization value

The value you want the global variable to be initialized with. This parameter is optional.

Example:

```
Integer a = 12
```

This will create an Integer Variable named „a“ and initialize it with the value 12.

4.2.1.2 Declare

The declare section is used to instance Function Plug-ins. This instancing is done at parse time. So any initialization done in the constructor should be taken into account.

The syntax is as follows:

<plug-in type> <alias>[(init value1, init value 2, init value x)]

4.2.1.2.1 Plug-in Type

The type of the plug-in to instance

4.2.1.2.2 alias

The alias the Plug-in is to be known as in the body section.

4.2.1.2.3 initialization values

The values to initialize the plug-in with. Refer to the plug-ins documentation for details on the values to use.

Example:

```
PlgTable table("test1","test2")
```

This example will create a Table Object, and initialize it with the columns named „test1“ and „test2“.

4.2.1.3 Body

The body section contains the main program code of an olcus program. Most of this will probably consist of function calls, but these core commands are available:

4.2.2 Core Commands

The following section describes the basic commands implemented into the Body Section plug-in.

4.2.2.1 Conditionals

These commands will base the next executed instruction based on certain conditions.

4.2.2.1.1 If (Boolean condition), else, endif

The if command will execute the block of code between the if and the else instruction if the condition is „true“, or execute the block of code inside else and endif, if the condition is false.

4.2.2.1.1.1 Condition

The condition the IF command uses to decide what code to execute next. Needs to be of type boolean.

Examples of valid if commands:

```
If ( a > 12 )
  Console.echo(„too big“)
else
  Console.echo(„too small“)
endif

if ( a == 42 )
  console.echo(„the answer.“)
endif

if ( s == „r17“ ) goto speed;
```

4.2.2.1.2 while (condition), wend

The while command executes a block of code as long as a certain condition is met. As long as the condition is true, the execution is repeated.

Examples of valid while commands:

```
new(a:integer) = 20
while (a > 0)
  a = a – 1
wend
```

4.2.2.2 Exception Handling

Usually when an exception occurs, the program execution is halted. You can however, define that in case of an exception a certain set of instructions is executed, to allow the program to continue.

4.2.2.2.1 onException(label)

Instructs the program to jump to the specified label in case an exception occurs. This instruction can be called several times during program execution. The last set label will be valid.

4.2.2.2.2 Label

The label to jump to in case of an exception.

4.2.2.2.3 exception(message)

Raises an exception. Program execution will halt, unless an onException() instruction was used to redirect the error handler.

4.2.2.2.3.1 message

4.2.2.2.4 String: lastException()

Returns the message of the last exception that occurred.

4.2.2.2.5 label(alias)

The label Instruction marks a specific place in the code. The command itself does nothing, but it can be used as reference by other commands that allow jumping to a label.

4.2.2.2.6 alias

The Alias is a string containing only alphanumeric values. This alias needs to be provided to a goto or onException command, so the label can be identified.

Example:

```
label(beeblebrox)   #valid
label(„r17“)        #valid
label(go here)   #invalid
label(,,!&$!#)   #invalid
```

4.2.2.2.7 goto(label alias)

The goto command allows to continue execution at a specified label. Please notice that the goto command also allows the passing of a variable as label identifier, so the label to jump to does not need to be known at design time. However, you should make sure that the label you jump to actually exists, or an exception will be raised.

4.2.2.2.7.1 label alias

The alias of the label to jump to. Might also be a String variable, containing the valid alias.

Examples:

```
label(„r17“)
console.echo(„Hello world“);
goto(„r17“) # valid

label(„r17“)
console.echo(„Hello world“);
goto(r17) # valid

new (a:integer) = 16
a = a + 1
label(„r17“)
console.echo(„Hello world“);
goto(„r“+a) # valid
```

4.2.2.2.8 sub(alias, parameter1, parameter2, parameter3, ...)

The „sub“ command works somewhat similar to the label command, and can be used in a similar way, but offers a lot more possibilities.

4.2.2.2.8.1 Alias

The alias this subroutine will be known under. The alias is globally available, so the gosub command for this subroutine can be called from anywhere. The same restrictions as in the label command apply.

4.2.2.2.9 parameters

Parameters define what kind of information may be passed to the subroutine. A parameter is defined similar to the structure of a new() command. Parameters behave like Local Variables, and cease to exist whenever the end or return command is encountered.

<type>:<alias>

4.2.2.2.9.1.1 type

The variable type. Refer to the „global section“ reference for available variable types.

4.2.2.2.9.1.2 alias

The alias the parameter is to be known under inside of the subroutine.

Examples:

```
sub(add,a:integer,b:integer)
  new (d:integer) = a + b
return(d)

sub(simple)
 console.echo(„This is rather simple“)
return
```

4.2.2.2.10 gosub(alias, parameter1, parameter2, parameter3...)

The gosub command will allow to call a subroutine. The commands inside the subroutine will be evaluated until a return command is encountered. Then the execution will continue after the gosub command.

4.2.2.2.10.1 alias

The alias of the subroutine to execute, also accepts a String variable.

4.2.2.2.10.2 parameters

The parameters to be passed to the subroutine. Please keep in mind that the parameters of the subroutine need to match the passed parameter numbers and types. Overloading is not possible.

Examples:

```
b = gosub(add,17,42)

gosub(simple)
```

4.2.2.2.11 new(alias:type)

Creates a local variable, valid inside of the current subroutine. The Variable ceases to exist whenever a return or end statement is encountered

Keep in mind that the code between the body and the end statement is also handled as a subroutine, so you can use local variables there, too, as needed.

4.2.2.2.11.1 alias

The alias the variable is to be known as in the subroutine.

4.2.2.2.11.2 Type

The variable type. Refer to the global section documentation for available types.

Examples:

```
new (a:integer) = 42
```

4.2.2.2.12 integer: fork

The fork command allows to fork program execution into two independent threads. The program code can identify the thread it is running in by evaluating the integer value returned to by the fork command. The original thread will find the integer value 0, while the newly created executor thread will have a return value of 1. Please keep in mind that concurrent access to variables in this context might be problematic.

Example:

```
if (fork == 1)
  new(a:integer) = 20;
  while (a > 0)
    console.echo(„thread 1 : „+a)
   a = a - 1
  wend
else
  new(b:integer) = 15;
  while (b > 0)
    console.echo(„thread 2 : „+b)
    b = b - 1
  wend
endif
```

4.2.2.3 Operators

Olcus supports various operators for simple math operations. For more complex operations please refer to math base function plug-in, that is instanced by default for every olcus program.

If two variables of different variable type are used with an operator, the resulting variable type will the the one that is determined as the „stronger“

4.2.2.3.1 '=' (Assignment)

This operator is used to assign a value to a variable. The syntax is

<target alias> = <source variable>

4.2.2.3.1.1 target alias

The destination variable

4.2.2.3.1.2 source variable

The source variable, or constant.

Example:

```
a = 42   # Assigns the variable 42 to a
new(a:integer) = 17 # Assigns the value 17 to newly created local variable a
```

4.2.2.3.2 '' (Multiplication)*

Multiplies two operands.

<operant1> * <operant2>

4.2.2.3.3 '/' (Division)

Performs a division

<dividend> / <divisor>

4.2.2.3.4 '+' (Addition)

Adds operant2 to operant1

4.2.2.3.5 '-' (Subtraction)

Subtracts operant2 from operant1

<operant1> * <operant2>

Examples:

```
a = 12 + b * 3
```

4.2.2.4 Comparators

Comparators allow to compare two values in different ways. Some comparators might be phrased in different ways so you can feel right at home from your current programming language. In comparison to normal operations, comparators always return a boolean value.

4.2.2.4.1 '==' (Equals)

Performs a comparison between the two operators, and returns 'true' if both variables are equal.

4.2.2.4.2 '<>' or '!=' (Does Not Equal)

Performs a comparison between the two operators, and returns 'true' if both variables are not equal.

4.2.2.4.3 'and' or '&' or '&&' (logical and)

Compares two boolean operators, if both operators are true, true is returned, otherwise false.

4.2.2.4.4 'or' or '|' or '||' (logical or)

Compares two boolean operators, if one of the operators is true, true is returned.

4.2.2.4.5 'xor' or '^(logical xor)

Compares two boolean operators. If one of the operators is true and the other is false, true is returned otherwise false.

4.2.2.4.6 '>'

Returns true if operator 1 is bigger than operator 2

4.2.2.4.7 '<'

returns 'true' if operator 1 is smaller than operator 2

4.2.2.4.8 '>='

Returns 'true' if operator 1 is bigger than operator 2, or if operator 1 equals operator 2

4.2.2.4.9 '<='

Returns 'true' if operator 1 is smaller than operator 2, or if operator 1 equals operator 2

4.2.2.4.10 'not' or '!'

The not command will invert a boolean value. True will become false, and false will become true.

Examples:

```
if (!(a > 3) || (b < 12))
```

4.2.2.5 Comments

Comments should be included in the source code so code readability is improved. Comments are marked by the '#' sign. All characters following a '#' character will be ignored by the parser, until a new line is reached. Since a line does not need to contain code, commenting whole lines is possible.

Comments work in all sections.

Example

```
# This is strange
new (a:integer) = 42    # This assignes the 42
```

4.2.2.6 #Include(targetpath)

The include command will include another character stream exactly at the point where the include is placed. The effect is equal to actually cutting and pasting the text referenced by the include instruction. The include indeed needs a preceding '#' commend sign to work correctly. This is the only exception where the comment identifier will be ignored.

Include works in all sections.

Example

```
#include(„c:\commonroutines\useful.odn“)
```

4.2.3 Standard plugins

There are a few plug-ins that automatically get instanced for Every olcus program. These plug-ins provide static functions, so its not necessary to instance them again, but if you must, you can. They are listed in the plug-in manager.

4.2.3.1 System
Contains system default functions, that are useful to most programs.

4.2.3.1.1 system.sleep(miliseconds: integer)
This suspends the thread for the number of miliseconds provided.

4.2.3.1.1.1 Miliseconds
The number of miliseconds to suspend thread execution.

4.2.3.1.2 system.msgbox(msg: string)
Displays a message box, and suspends execution until the user confirms the message.

4.2.3.1.2.1 Msg
The message to display in the message box.

4.2.3.1.3 String: system.inputbox([title])
Creates a text field box, and waits for user input. Thread execution will resume as soon, as the ok button is clicked, and the text will be reported as an exit value.

4.2.3.1.3.1 title
the title of the input box. This parameter is optional.

4.2.3.1.4 integer: system.militime()
Returns the number of miliseconds passed since system startup.

4.2.3.1.5 String: system.dateTime()
Returns the current date time stamp

4.2.3.2 Console
The console plug-in offers some methods for communication with the console

4.2.3.2.1 console.echo(msg: string)
Displays a message String in Standard output.

4.2.3.2.1.1 Msg
The message to be send to the console

4.2.3.3 Math
The math plug-in contains functions that have to do with mathematical operations.

4.2.3.3.1 math.random(range:integer)
Returns a random number inside of the specified range. The least value will be 0, the maximum value is the one specified.

4.2.3.3.2 math.maxInt()
Returns the maximum value allowed for an integer variable

4.2.3.3.3 math.minInt()
Returns the minimum value allowed for an integer variable

4.2.3.3.4 math.sin(gradient:integer)

Performs a sinus operation.

4.2.3.4 Table(column name1, column name2, ...)

The table plug-in allows the user to create a table window, to display data. The table model is based on rows and columns. Columns can be created during initialization or during program execution. Rows need to be added before the columns of a row can be filled with values. Refer to the table example program for reference.

During initialization you can provide a set of column names that will then be used to create columns as soon as the table is displayed.

4.2.3.4.1 Table.setTitle(title:string)

sets the title of the table window, for better reference should your application require more than one table to display data.

4.2.3.4.2 Table.setPos(x:integer, y:integer)

sets the position of the window. The position is provided in pixels, relative to the upper left corner of the display.

4.2.3.4.3 Table.addColumn(name:String)

creates a column for the table object. The column title will be set by the name provided. Keep in mind that names can also be used as references, so column names should be kept simple.

4.2.3.4.4 Table.addRow()

This adds a row to the table. AddRow must be called at least once so that data can be displayed in the table. Without a row no data can be written.

4.2.3.4.5 Table.setRow(index:integer)

Sets the row, indicated by the index parameter. Please make sure that the row exists before referencing it.

4.2.3.4.6 setValue(column:[integer]/[string], value:String)

Sets a value for a column in the currently active row. The column can either be referenced by an integer index starting with 0, or the columns name.

4.2.3.4.7 setColSize(column:[integer]/[string], size:integer)

Sets the size of a column. The column is either referenced by an integer index or the columns name provided in addColumn(). The column's width will be set to the number of pixels provided by the size parameter.

4.2.3.4.8 autoSizeCol(column:[integer]/[string])

Automatically sizes the column based on the size of the columns name. The column can either be referenced by an integer index, starting with 0, or the name of the column.

4.2.3.4.9 autoSize()

Automatically sizes all columns in the table, based on the size of the column name.