Olcus.net Reference manual

Version 3.2.3578
11/2009

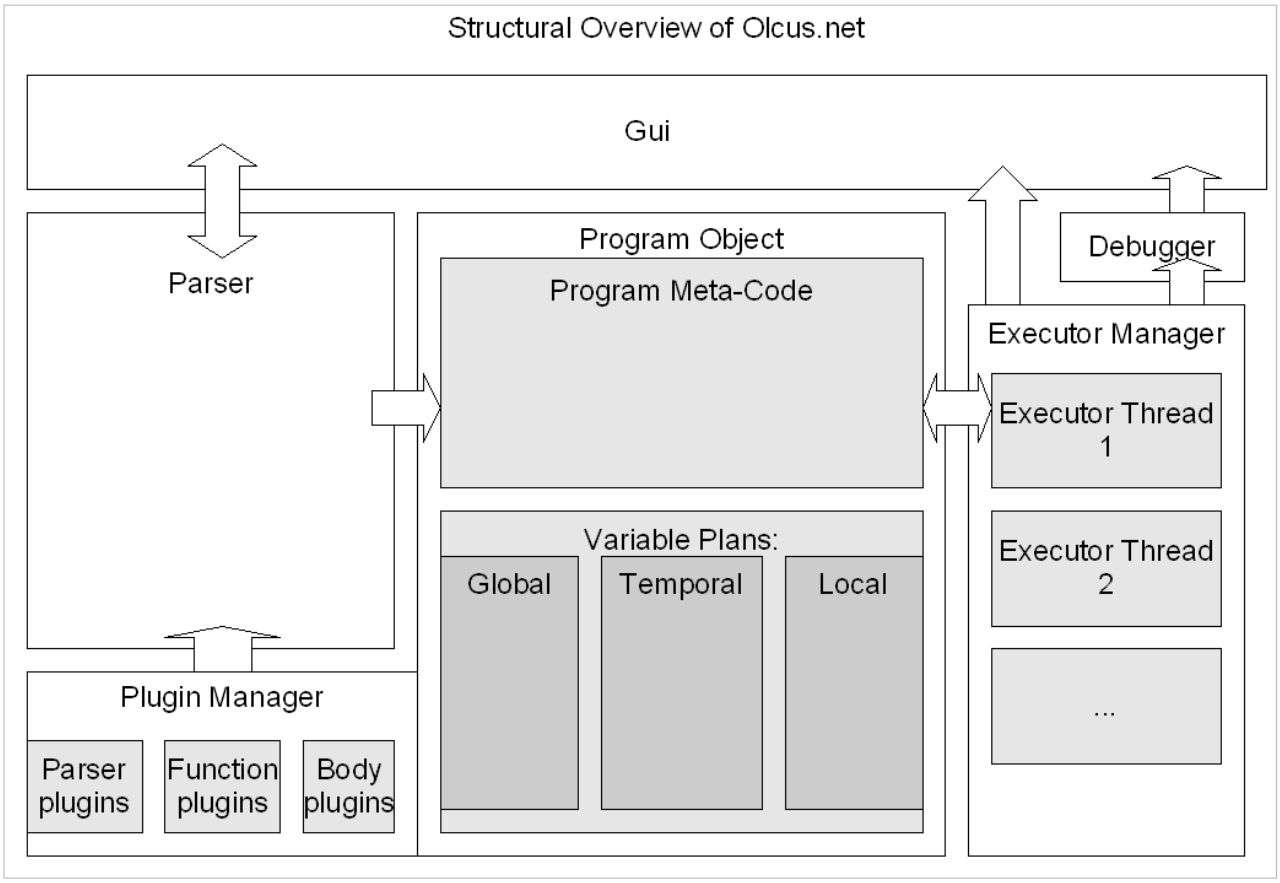
Index

Aims

Olcus.net or Olcus 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 witout need to touch any of the core components.
- Extendability: Olcus.net is designed with several Plugin interfaces that allows 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 plugins will not be granted unless plugins are evaluated and signed.

Structural Overview



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

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.

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.

Meta Code

Olcus meta coda consists of an arraylist 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 plugin development section of this document.

Variable Plans

Variable plans need to be used because at parsetime 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 ArrayLists for three different variable Types:

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)

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.

Temp

Temporal variables are automatically created by the parser to allow MicroCode commands to pass values. Temporal variables will store inbetween 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.

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.

For 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
```

Internal

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

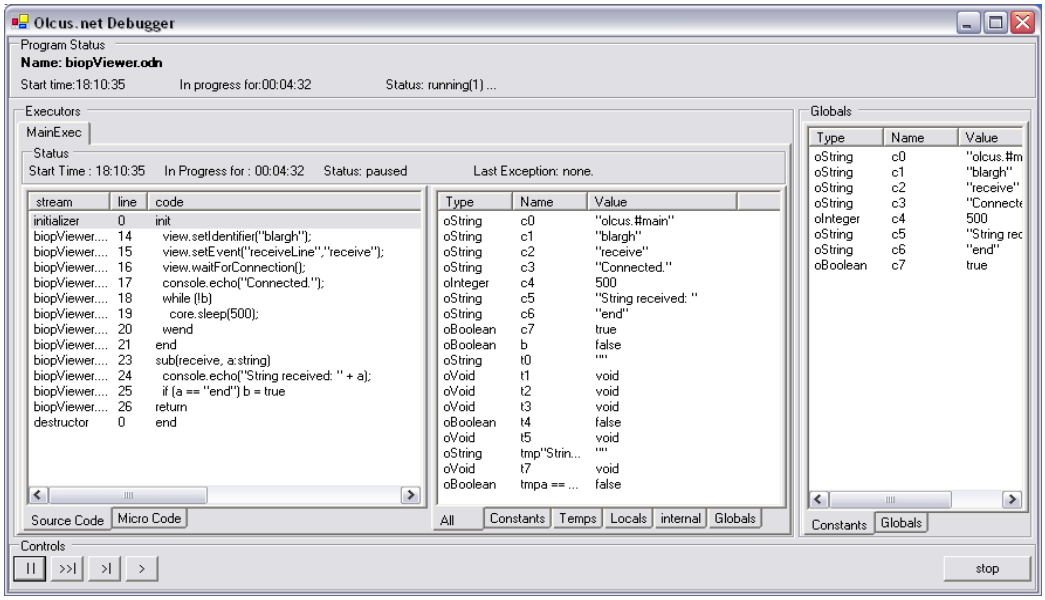
Executor Manager

The exectuor 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 debuggin session.

Executors

The executor represents a single thread of Program execution. The executor object contains an arraylist 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.

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.

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.

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 linenumbers inside the code) and finally the column „code“ contains the character stream the parser used to create the microcode from.

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 sourcecode the meta command was created of.

Variables

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

All

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

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.

Temps

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

Locals

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

Internal

This tab shows a string representation of all currently internally stored objects. (Usually stored for internal communication by plugins) 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 plugin, inside of a .net ide.

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.

Controls

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

'||' Pause

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

'>>|' Single Line Step

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

once.

'>|' Single meta Command

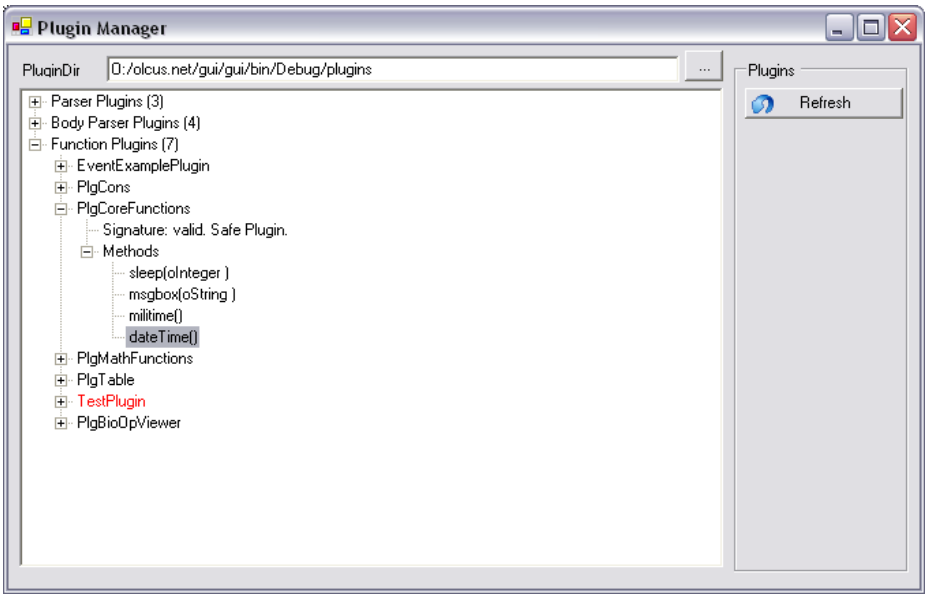
Executes only one Meta-Command at a time, then pauses execution again.

'>' Play

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

Plugin Manager

The plugin manager can be used to gain an overview of all plugins currently available to the olcus.net application and their status.



Function Plugins

The node function plugins contains information about all plugins currently extending olcus' function calls. Each plugin node contains information about available methods, their parameters, and plugin events. This can be useful during code creation, for easy access to keywords neccessary for making succesful functioncalls.

Section Plugins

This node contains information about all currently installed section plugins.

Body Section Plugins

This node contains information about all installed body section plugins and their status.

Services

The maintenance object offers several useful services.

Logger Service

The logger service is a plugin based central logging service.

Outputters

Logging messages can be routed into various destinations, using OutPutter classes. Several OutPutter classes may be stacked to direct the Stream into various destinations at once. Each outputter plugin can decide on its own if the messages priority is relevant to it.

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 logfile will be recreated.

LogOutToConsole

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

LogOutToListBox

This will add messages to a Listbox provided to the constructor of the Outputter.

LogOutToTextBox

This will add messages to a TextBox provided to the constructor of the OutPutter. In contrary to the listbox outputter one message my occupy serveral lines.

Priorities

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

The following loglevels are available, ordered by their priority level:

Logger.DEBUG

Messages used during the debugging state of developing plugins.

Logger.INFO

Non Vital Information about the program execution

Logger.WARNING

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

Logger.ERROR

Errors that occur. Usually at least this loglevel should be set.

Logger.CRITICAL

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

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 hardcoded 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.

Code Development

This section will give an overview about code development in Olcus.net. You will find many similarities to other programming lanugages, which should make it easy to adapt to.

Code Structure overview

Olcus code is structed 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.

Sections

Available sections are:

Global

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

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

type

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

alias

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

Initialization value

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

For example:

Integer a = 12

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

Declare

The declare section is used to instance Function Plugins. This instancing is done at parsetime. So any initialization done in the constructor should be taken into account.

The syntax is as follows:

<plugin type> <alias>[(init value1, init value 2, init value x)]

Plugin Type

The type of the plugin to instance

alias

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

initialization values

The values to initialize the plugin with. Refer to the plugins 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“.

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:

Core Commands

The following section describes the basic commands implemented into the Body Section plugin.

Conditionals

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

If (boolan 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.

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;
```

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
```

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.

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.

Label

The label to jump to in case of an exception

exception(message)

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

Message

The message contained in the exception

String: lastException()

Returns the message of the last exception that occured.

Helpers

inc(number)

increases the value contained in the referenced variable by 1.

dec(number)

decreases the value contained in the refrenced variable by 1.

Program Pointer Control

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.

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
```

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.

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
```

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.

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.

parameters

Parameters define what kind of information may be passed to the subroutine. A parameter is defined similiar 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>

type

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

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
```

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.

alias

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

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)
```

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.

alias

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

Type

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

Examples:

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

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 itneger value returned to by the fork command. The orginial 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
```

Operators

Olcus supports various operators for simple math operations. For more complex operations please refer to math base function plugin, 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“

'=' (Assignment)

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

<target alias> = <source variable>

target alias

The destination variable

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

'*' (Multiplication)

Multiplies two operants.

<operant1> * <operant2>

'/' (Division)

Performs a division

<divident> / <divisor>

'+' (Addition)

Adds operant2 to operant1

'-' (Substraction)

Substracts operant2 from operant1

<operant1> * <operant2>

Examples:

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

Comperators

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

'==' (Equals)

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

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

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

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

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

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

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

'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.

'>'

Returns true if operator 1 is bigger than operator 2

'<'

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

'>='

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

'<='

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

'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))
```

Comments

Comments should be included in the sourcecode so codereadability 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
```

#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“)

Standard plugins

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

System

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

system.sleep(miliseconds: integer)

This suspends the thread for the number of miliseconds provided.

Miliseconds

The number of miliseconds to suspend thread execution.

system.msgbox(msg: string)

Displays a message box, and suspends execution until the user confirms the message.

Msg

The message to display in the messagebox.

String: system.inputbox([title])

Creates a textfield 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.

title

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

integer: system.militime()

Returns the number of miliseconds passed since system startup.

String: system.dateTime()

Returns the current date time stamp

Console

The console plugin offers some methods for communication with the console

console.echo(msg: string)

Displays a message String in Standard output.

Msg

The message to be send to the console

Math

The math plugin contains functions that have to do with mathematical operations.

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.

math.maxInt()

Returns the maximum value allowed for an integer variable

math.minInt()

Returns the minimum value allowed for an integer variable

math.sin(gradient:integer)

Performs a sinus operation

String

The string plugin defines a set of useful operations for the handling of String variables. It is implemented as the object „string“ by default. You can instance other String plugin objects, but it is usually not neccessary.

string.sub(source:String, start:integer, end:integer):string

this method returns a subsection of the provided string. For example:

string.sub(„hamburger“,4,8) will return „urge“

source

The string to return the subsection of.

start

the startindex of the string to return.

end

the endindex of the string to return.

string.subbylen(source:String, start:integer, len:integer):string

This method reutrns a subsection of the provided string. In comparison to the „sub“ method, this method will not need an end index, but the total lenght of characters to be extracted.

string.sub(„hamburger“,4,4) will return „urge“

source

The string to return the subsection of

start

the startindex inside of the sourcestring

len

the number of characters to return from the startindex.

string.left(source:string, len:integer):string

Returns a number of characters starting from the left side of a source string.

source

The String to return characters of.

len

The number of chars to return

string.right(source:string, len:integer):string

Returns a number of characters starting fromt he right side of a string.

source

The String to return characters of.

len

The number of characters to return.

string.indexof(source:string, sub:string):integer

Returns the position of a substring inside of another string. If the substring is not found, -1 will be returned. e.g. string.indexof(„supercalifragilisticexpialigothic“,“frag“) will return 9.

source

The string that contains the substring.

sub

The substring to locate.

GUI

The gui plugin can be used if an olcus.net script needs to be configured by the user. The plugin can then be used to create an interface that provides an easy way to review and change configuration values.

The gui object consists of a window that is organized into tabs. By default one window has only one tab. If you want to organize your interface, more tabs can be added and manipulated easily.

Each Tab can contain one or more controls that can be used for user interaction. Each control has at least an ID it can be identified by (and therefore has to be unique inside of the gui element) and a value that can be read or set.

Manipulating the gui window

The main gui window can be manipulated using the following methods

show()

when created the gui window is hidden. Once you have placed all controls, you should use the show method so the window is displayed.

hide()

once you are done with this configuration dialog, you can hide it for later use.

setWindowSize(width:integer, height:integer)

This sets the dimensions for the gui window.

setWindowPos(x:integer, y:integer)

This allows to position the window, relative to the top left corner of the screen.

setTitle(title:String)

This method allows to change the title shown in the title bar of the gui window

Tabs

Tabs can be manipulated using several methods. Like controls, every tab has a unique id, which will be used to reference it.

addTab(id:string,[text:string])

adds a tab to the current gui window. The tab will be referenced in followup commands by the provided id. The text parameter is optional. If text is provided it will be used as the text shown to the user in the tab. If it is not provided the ID will be shown to the user as tab title.

The first added tab will actually overwrite the default tab, so to have two tabs, you will have to use the addTab function twice.

setTab([id:String]/[index:integer])

activates a tab referenced either by the tab's id, or an index starting with 0 for the first tab.

All methods that add controls will add automatically to the currenty active tab.

Adding Controls

There are various methods to fill an active tab with the user controls. Every control has a unique id that will be used by methods to indicate said control. Every Control also has a value that can be used to query the controls current state. The text attribute is shown to the user.

On adding controls will be placed onto the tab in the order they were created. Controls can be placed freely using the setSize, setPos, setHooks methods.

addTextBox(id:String,text:String, value:String)

this adds a textbox to the currently active tab, that allows the user to enter a textstring. Which can then be found in the value attribute.

addCheckBox(id:String,text:String, [value:String] / [value:boolean])

this adds a CheckBox to the active tab. The value parameter can either be provided as a string, which has to be „true“ or „false“, or as a boolean parameter. If anything incompatible is passed as a value, and exception will be raised.

addButton(id:String, text:String)

this adds a button to the active tab. A button does not have a value.

addSeperator([text:String])

adds a seperator to the currently active tab. If a text portion is provided it will be displayed as the title of the seperator. Can be used to organize your tab into categories.

addRadioGroup(id:String)

adds a Radio group to the currently active tab. A radio group is a set of options of which only one can be selected by the user. In its initial state the radiogroup does not have any options, so this control only makes sense if you use addRadioGroupOption at least twice.

addRadioGroupOption(id:String, id:String, [text:String])

adds an option to a Radio Group. The text argument is optional, if it is not provided the id will be shown to the user instead. Use the setValue method to select which option you want select as default. (setValue(„radioid“,“optionid“))

Manipulating controls

Several methods allow you to manipulate existing controls. Controls are always referenced by the id

you provide to the add Method.

setPosition(id:String, x:String, y:String)

Sets the position of the control, relative to the top left corner of the tab the control resides in. This method will deactivate all autosizing options of the referenced control. Autosizing will have to be reenabled using the setHooks method.

setSize(id:String, w:String, h:String)

Sets the size of the indicated control. This method will deactivate all autosizing options of the referenced control. Autosizing will have to be reenabled using the setHooks method.

setHooks(id:String, top:boolean, right:boolean, bottom:boolean, left:boolean)

Sets the autosizing options for the indicated control. If you set a hook to true, that part of the control will stay attached to the corresponding part of the gui window, so the controls change size along with the gui changing size.

Events and Control

The gui plugin defines one convinience method to wait for a button being pressed, and one button event, that you can use to react to user input.

event genericButtonPressed(id:String)

the genericButtonPressed event is fired whenever a button is pressed. The buttons id will be passed to the called subroutine, so you can determine which button raised the event and react accordingly. Refer to the example script gui.odn in the examples directory.

waitForButton(id:String)

This method will pause script execution until a certain button is pressed. This could be achived inside of the script as well, this method however should be much more convinient.

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

The table plugin 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.

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.

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.

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.

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.

Table.setRow(index:integer)

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

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.

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.

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.

autoSize()

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

Plugin Development

Olcus extendable architechture depends on the development of plugins, either by FBI-Science or 3rd party companies. This documentation should be part of the System Developers Kit.

To create a new Plugin you need to follow the following steps:

1. create a new Class-Library project
2. add the class library „core“ to the references section.

The language used to develop the plugin is more or less irrelevant as long as its a .net conform language, like for example C# or vb.net.

Olcus Variable Types

Before the details of plugin developement will be discussed, a quick overview of olcus variable system is neccessary.

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 plugin function can work with the contence. The get function returns the natural basetype, no override will be neccessary.

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

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 convinience the oVoid Type has a static method called getVoid() which returns a freshly instanced oVoid variable that can be returned to the olcus engine.

Function Plugins

A function plugin extends the functionality of the Olcus engine. Methods defined inside of a Function Plugin class will be directly transparently available to the olcus program code.

For a Function plugin to be recogniced the Type (class) will have to be Inherited from plugins.Plugin

functions

A few requirements have to be met by a method to be automatically exposed to the plugin mananger:

1. All methods that are to be exposed as plugin functions need to consist of olcus variable types only. That includes void methods. These need to return the oVoid element. This is a security measure to ensure that the olcus engine can handle all included parametertypes.
2. Exposed methods need to be public so the olcus execution engine can reach the method from outside.
3. An olcus function plugin method always needs to return a value, even if that value is only oVoid.

Once a method is created and all parameteres are valid olcus variable types, it will automatically show up in the plugin manager and will be available to calls from the programcode.

Examples:

```
private void thisDoesNotWork(oInteger a) {   #invalid
}

private oVoid thisDoesWork(oInteger a) {   #valid
  return oVoid.getVoid();
}

private oVoid thisDoesNotWork(oInteger a, String b) { #invalid
  return oVoid.getVoid();
}

private oString thisDoesWork(oInteger a, String b) { #valid
  return new oString(b.get() + a.get());
}
```

Function methods are overridable. That means that a function might exist with the same name as long as the parameter signature defers. That does not apply for the return parameter.

Examples:

```
# Valid overriding
private oVoid test1(oInteger a) {}
private oVoid test1(oString s) {}
private oVoid test1(oString s, oInteger a) {}

# invalid overriding
private oVoid test1(oInteger a) {}
private oInteger test1(oInteger a) {}
private oString test1(oInteger a) {}
```

If you need to have methods with different return types it would be recommended to simply add the returntype inside of method name.

```
# Valid
private oVoid test1asVoid(oInteger a) {}
private oInteger test1asInt(oInteger a) {}
private oString test1asString(oInteger a) {}
```

initializers

Initializers allow to pass parameters to the Plugin at creation time. Initializers are defined inside of the Declare section. Initializers can be overriden, different paremeter signatures will automatically select the best fitting initializer.

The return type of initializers will be ignored. The method needs to be public, and literally called „initializer“ to be recognized as an initializer method. That method will not be exposed to the the rest of the program and can only be called from the „declare“ section of the program.

The same as normal function methods, initializers are overridable. The initializer used will be automatically selected by the best fit of available command parameters.

Examples:

```
void  initializer(oInteger i1, oInteger i2) {    // Can be called from inside the program as
}                                                // Declare
                                                 // Pluginname alias(17,42)
```

If the number of parameters is unknown at design time, there is an alternative way of specifying an initializer:

```
void initializer(ArrayList lst) {
}
```

if this type of method declaration is available, and none of the other initializers fits the given parameters it is called instead. The number of parameters will be passed as an ArrayList of oVariables. The initializer function needs to take care of checking the parameter types, the number of parameters, and the validity of the contence.

Creation / Destruction

Plugin types (classes) offer a lot of overrideable methods used for initialization and deinitialization.

create

Create is called upon creation of the plugin, and has to do all initializations that cannot be done inside of the constructor.

init

This method is called in every instanced plugin object just prior to program execution. It can be used to create a „clean“ state of the plugin, if the program is executed several times in a row, without being parsed again. (which should be the default way programs are used, so the init method should ensure to clean up correctly)

stop

This method is called in every instanced plugin, once the program execution ends. This can be used to close connections, stop receiving data, or whatever the plugin needs to stop once the program is not running anymore. (keep in mind that the .net enviroment will continue to function even if no olcus program is in progress)

destroy

This method is called whenever the olcus enviroment is shutdown, or a new parse process is started, just prior to destruction of the instanced plugin object. All critical dinitialization code needs to be placed in here. (freeing up sockets, closing files, ect.)

events

The olcus engine supports the raising of events, raised by Plugin object. An event will basically create a new Executor Thread, and then execute a subroutine previously indicated by the program code. Therefore command parameters can be passed to the subroutine.

An event is handled by an object, representing said event. Event objects should be created inside of the object's constructor. While event objects created outside of the constructor will work, they will not show up in the plugin manager, and will make debugging difficult.

The plugin object should keep a link to a created event object so the event can be raised. There are two ways of raising an event.

The subroutine to be called in case an event is raised will be set in the program code using the .setEvent() method, that is inherited by every plugin object.

Plugin.createEvent(name)

creates an event object. The resulting object of the type Event, should be stored somewhere, so you can use it later to raise the event.

Plugin.setEvent(<eventname>,<subname>)

Determines at runtime what subroutine is to be called in case the an event is raised.

Eventname

The name of the event, provided as parameter by the

subname

The name of the subroutine to be called in case the event is raised. Please document carefully what kind of paramters the subroutine needs to be able to accept. If the provided and expected parameters do not match, an exception will be created at runtime, since the parser is unable to determine what paremeters are expected by the event.

event.raise()

If no parameters need to be passed to the event it is sufficient to just call event.raise. The olcus engine will then automatically create an executor thread and start at the designated subroutine.

If setevent was not called inside of the program before the event is raised, the event raising will be ignored.

event.raise(ArrayList parameters)

Raises an event and passes the parameters for the used Subroutine via an arraylist containing

oVariable objects. The order of the parameters and types need to fit to the order of parameters and parameter types expected by the subroutine, or an exception will be raised at runtime.

event.setExpectedParameterTypes(ArrayList types)
This function call can be used to create parsetime warnings when the set of parameters expected by the event do not match the expected parameters at runtime. This will make it easier to debug code that uses events that are parameter driven.
There are two alternatives to implement this securety measure:

Examples:

```
ArrayList types = new ArrayList(); // Most clearly structured
types.add(typeof(oString));        // Makes the event expect a sub, that
types.add(typeof(oString));        // Expects two strings as paramters.
Event.setExpectedParameterTypes(types);

// The same as above as a one-liner
event.setExpectedParameterTypes(new Type[]{typeof(oString), typeof(oString)});

// The same as the two examples above, integrated in the create event statement.
Event myEvent = createEvent(„MyEvent“,new Type[]{typeof(oString),
typeof(oString)});
```

context objects

Context objects are used to allow a method access to the context it is running in. The context is the current Program the Plugin is running in, the Current Executor that has just called the methods, the local Variable Space, and the local internal Variables.

A method can request a context object, by specifying its first paramter to be of the type core.Context. The method calling mechanism of the olcus engine will automatically recognice the context object, and provide it. The requested object will not be visible to the parser.

Example:

```
// plugin method:
  oVoid doSomethingInContext(Context con, oString str) {
      // ... code ....
  }

# Olcus function call:
  somePlugin.doSomethingInContext(„Hello“)    # Valid, because context objects
                                               # are not visible to the parser
```

The context object's getters and setters and methods can be used to provide access to the local context to the plugin.

This will only be neccessary in exceptional cases. The Table Plugin for example, uses the context object to store the current TableRow that is assigned to the local Executor Thread. Every thread will get its own table row, when new line is executed, so that interthread access problems won't occur.

internal variables

Internal Variables are used to store information that needs to stay in thread context. This mechanism is used in the Table Plugin example stated above. It contains of two very simple method calls that can be used to store and retrieve information in a context safe way.

setLocalInternal(String name, Object obj)

Stores an object into the internal context-safe repository. Please choose a secure, unique name, so that you will not interfere with other plugin's storage here. It is recommended to create a name in this manner: <pluginname>_<objectname> to minimize the danger of accidental object replacements inbetween plugins.

Object getLocalInternal(String name)

Retrieves an object from the local Internal Storage repository. The object is identified by the name passed to the setLocalInternal. Casting of the object type will be neccessary.

bool existsLocalInternal(String name)

Returns true, if an object exists by the name provided to the method. Can be used to check if an object exists, because getLocalInternal calls to non existing objects will cause an exception.

License Checking

Each Plugin is free to define its own way of licensing. The parser will invoke the method **bool checkLicense()**. When you plugin does not require any licensing, then you can simple not override the method. The plugin will then always be licensed.

When you override the method, all your method has to do is return „true“ when everything seems to be valid, or raise an exception, explaining to the user why licensing was found to be invalid. The parser will then stop parsing the program code when instancing the plugin is attempted.

Command objects

Command objects will usually have to be created for Section Plugins or BodySection Plugins to do any useful work. A command object is what the „meta-code“ that the olcus engine executes conssists of. In essence the MetaCode Program is an arraylist containing objects inherited from Command objects.

A command object has to be inherited from core.commands.Command. There is one abstact methdo „execute“ that needs to be overwritten for the command to be able to do any work.

When designing a command object its very important that you keep in mind that the command object will likely be accessed from serveral threads at once, therefore no vital thread sensitive information may be stored inside of the command object itself. Therefore all used variables should be stored in oVariables accessed via the oPlan abstraction layer.

When designing a function plugin the MethodWrapper command will do this work for you, but its neccessary to do these conversionsteps yourself when you implement your own command modules. Learning by exmple should be easily enough though.

A very simple command object could look like this:

```
using System;
using core.variables;
using core.executer;

namespace core.commands
{
	/// <summary>
	/// Not command. Inverts a boolean variable
	/// </summary>
	public class cmdNot:Command
	{
		oPlan dest = null;   // The destination Variable
		oPlan o1 = null; // The source Variable

		/// <summary>
		/// Negates the content of a variable, and stores the result into another
		/// </summary>
		/// <param name="source">The source Variable</param>
		/// <param name="destination">The destination Variable</param>
		public cmdNot(oPlan destination, oPlan o1) {
			this.dest = destination;
			this.o1 = o1;
			this.name = "not";
		}

		/// <summary>
		/// Assigns the value of the source variable into the destination Variable
		/// </summary>
		/// <param name="exec">The context in which the command is running</param>
		public override void execute(Executor exec) {
			// Variable extraction
			oBoolean dst = (oBoolean)dest.retrieveVariable(exec);
			oVariable op1 = o1.retrieveVariable(exec);
```

```
                if (!op1.GetType().Equals(typeof(oBoolean))) {
                        oVariable tmp = op1;
                        op1 = (oVariable)oVariable.instanceClass(typeof(oBoolean));
                        op1.set(tmp);
                }
                oBoolean op1b = (oBoolean)op1;
                // Actual Calculation
                dst.set(!op1b.get());
        }
    }
}
```

The example is the NOT command, one of the more simple commands implemented in the Olcus engine.

We will discuss each section of the command to clarify how a command Object works:

Variables

```
        oPlan dest = null;   // The destination Variable
        oPlan o1 = null; // The source Variable
```

As you can see, a command usually requires at least a source and a destination variable link. You cannot store a reference to the variable itself, because at creationtime (during parsing) the variable does not exist yet.

Constructor

```
        public cmdNot(oPlan destination, oPlan o1) {
                this.dest = destination;
                this.o1 = o1;
                this.name = "not";
        }
```

In this case the constructor fills the local Variable links. That does not have to happen inside of the constructor, but it should be the most convinient way.

Also the Commands „Name“ is initialized here. This name is what shows up in the debuggers microcode display.

The oplans to use are provided by the parser method.

execute(Executor exec)

The execute method is called by the executor at runtime, when the command is evaluated. The executor that called the method is provided as parameter, so operations like extracting local varaibles from the context are possible. These things are mostly automated, but not fully.

```
        public override void execute(Executor exec) {
```

```
// Variable extraction
oBoolean dst = (oBoolean)dest.retrieveVariable(exec);
oVariable op1 = o1.retrieveVariable(exec);
```

As you can see, the destination variable is extracted from the oPlan link. That is done via the retrieveVariable method, that needs the current executor for the right variable to be returned.

```
if (!op1.GetType().Equals(typeof(oBoolean))) {
      oVariable tmp = op1;
      op1 = (oVariable)oVariable.instanceClass(typeof(oBoolean));
      op1.set(tmp);
}
oBoolean op1b = (oBoolean)op1;
```

When calling function methods, variable conversions will be done automatically. Inside of a command class however, conversions have to be done manually. Here you can see the type is checked, if the type is not boolean, a conversion is attempted via the .set method, that can convert one variable format into another, when used on the target variable.

```
// Actual Calculation
dst.set(!op1b.get());
```

When we are sure that every variable is in the right format, the actual work can be done. Its quite simple in this case. The destination variable is loaded with an inverted source variable.

Command Creation

Command objects are usually instanced by a parser plugin. That might be a section plugin, or a body section plugin, both do the instancing in the same way.

```
public oPlan doException(String line) {
      // Parse Parameters
      ArrayList conds = bodypars.parseParameters(pars.extractParams(line));
      if (conds.Count != 1)
         throw new Exception("Invalid number of parameters");
      oPlan msg = (oPlan)conds[0];
      // Add Command to command list
      prog.addCommand(line,new cmdException(msg));
      return msg;
}
```

Shown here on the example of the doException function inside of a bodyParserPlugin class, is shown how a command is created.

The method.parseParameters() allows an easy parsing of parameters provided. If your command receives its parameters in braces, you can use the pars.extractParams function to extract the contence of the braces.

Parse parameters returns a set of oPlan objects, which will automatically be already evaluated. (Recursive command calls will already have been added to the stack. Therefore it is important to

add your own command AFTER parsing of the parameters)

You are then advised to theck the number of parameters, and the parameter types for compatability.

Then you can create the Command. That is done normally by its default constructor.

A command has to be added using the .addCommand method of the program object, so important links can be automatically established.

Section Plugins

A section plugin creates a new type of section for olcus source code. The way the characters inside such a section are interpreted is completely up to the plugin, the only limitation is that lines have to be seperated by semicolons, and that # characters handle remarks. The rest is completely up to the plugin. A section plugin activates automatically as soon as it becomes available to the olcus application. (It will show up in the Plugin Manager)

A section plugin has to be inherited from core.parser.ParseSectionPlugin.
Several methods are available for overriding. The parse method, of course, has to be overridden, for the plugin to be able to do work.

Constructor

The only thing that has to be done inside of the constructor is setting the keyword. The keyword activates a section plugin. If the parser encounters a keyword, the section with the fitting keyword will be activated. Choose the keyword carefully since it has highest priority inside of the parser architecture.

```
public ParseSectionTest()
{
  this.keyword = "testsection"; // Keyword
 }
```

Init

This method should be used for initialization. If you want to establish links to other plugins that has to be done here, since at creation time the other plugin might not exist yet.

If no initialization has to be done, you can simple choose not to override the init method. It is not mandatory.

Inbetween plugin communication

Plugins often need to exchange data. Since they are plugins you can not hardlink to them, since you cannot be sure the plugin will be there. Therefore a simple mechanism is implemented into the parser object, enabling a plugin to probe for other plugns.

Every sectionPlugin contains two links that will automatically be filled with the correct values.

„prog“ and **„parse“**

„prog“ is a connection to the program the parser is currently building. When you want to add a command or variable, you will need this link to the program object.

„parse“ is a link to the currently active parser object. The parser object offers the method „getSectionPlugin(name)“ that can be used to locate another section plugin. The returned link

should be locally saved, to improve perfomance. Usually you will want a lookup construct like this:

```
private ParseSectionGlobal globalVars = null;
/// <summary>
/// Returns the Global Variable plugin, so we can
/// access any global variables from within the body
/// </summary>
/// <returns>Returns a link to the globalSection plugin</returns>
private ParseSectionGlobal getGlobals() {
	if (globalVars == null)
		globalVars = (ParseSectionGlobal)pars.getSectionPlugin("global");
	return globalVars;
}
```

that way the plugin will only be fetched if it is needed, and later references can instantly return the link.

satisfyCommands

Many commands need a second pass to be filled with all parameters. We will use the goto command as an example. By the time the parser encounteres the goto command it is very likely that the label the command is to goto is not parsed yet. That is where satisfyCommands is useful. Once the parsing process is complete, the parser will call all satisfyCommands methods of all plugins. The plugins can then examine the microcode for commands that still need their parameters completed. The goto command then can safely locate its corrisponding label, and everything will be ready for execution.An example from the body parser Plugin:

```
/// <summary>
/// Cleans up all commands that need post parsing cleanup
/// </summary>
public override void satisfyCommands()
{
	foreach(Command cmd in prog.getCode())
	{
		if (cmd.GetType().Equals(typeof(cmdGoto))) ((cmdGoto)cmd).initialize();
		if (cmd.GetType().Equals(typeof(cmdGosub))) ((cmdGosub)cmd).initialize();
		if (cmd.GetType().Equals(typeof(cmdSetEvent))) ((cmdSetEvent)cmd).initialize();
		if (cmd.GetType().Equals(typeof(cmdOnException))) ((cmdOnException)cmd).initialize();
	}
}
```

oVoid parse(String line)

The parse method of the currently active Section plugin will be invoked for each line of code encountered. The oVoid returned is usually ignored, but since this is a recursive algorythm every parse method needs to return an oPlan object.

Body Section Plugins

A body Section plugin allows to extend the Body Section. Since the main part of the program code usually consists of a body secion, is is often useful to add new functionality to the section itself. Keep in mind that you may not use keywords already parsed by other parsing modules, or results might not be predictable.

A body Section Plugin has to be inherited from core.parser.BodyParserPlugin. The setup is similar to a section plugin. The constructor does not have to do anything specific, the initialization code in

there is mostly up to the programmer.

Init

The init method can be overridden for initialization code that cannot happen inside of the constructor. The parser will will automatically invoke each init method when the body section plugins init method is invoked.

oVoid parse(String line)

The parse method of all plugins are called in sequence and the currently line to interpret is passed. Once a parse method returns anything other than null, the parser will assume the line to have been correctly parsed, otherwise the next parserPlugin is given a chance to interpret the data.
The programmer needs to assure that anything besides null will only be returned if his method could completely interpret the line of code at hand.
If you can interpret the code but the paramters do not make sense in the context, it suffices to raise an exception so the parser will report the problem and stop interpreting the code.

```
public override oPlan parse(String line) {
        String tstLine = line.ToLower();

        // Do we have a command
        if (tstLine.StartsWith("sub")) return doSub(line);      // sub(name [, var:type, var:type, var:type]) (defines entry
point for subroutine)
        if (tstLine.StartsWith("gosub")) return doGosub(line);   // (ret)gosub(name,var,var,var)
        if (tstLine.StartsWith("return")) return doReturn(line); // return(retvalue)

        return null;
}
```

As you can see, you can easily use return statements to control the structure of your parser plugin. Whenever something interpretable is found, the return command will make sure that the parser does not look further. When null is returned in the end, the BodyParser Section plugin will try the next bodyParser Plugin, until no plugins are left. In that case an exception will be raised reporting „uninterpretable code“.

License Management

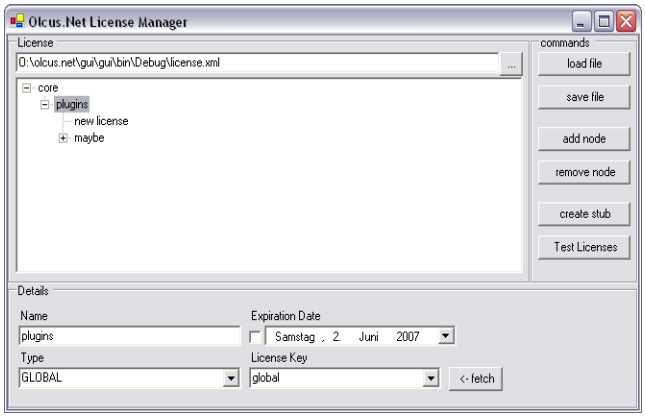
This section describes the license Management implemented into the olcus.net engine. Licensing is designed to be very flexible so various aspects of an application or plugin can be licensed using the same engine.

The licensing model is build into a tree structure. The root-node is always „core“. Core will be evaluated at program initialization. If core is invalid, no other olcus functionality will be enabled.

Every node required the parent node to be valid to be valid itself. (meaning that core.plugins.foo.bar will be invalid if „plugins“ is invalid.)

Every node can be attached to a different part of the system. There are also „global“ nodes that will work on every system, so you can organize your license nodes efficiently.

Signature files are created using the license manager:



The license manager is an editor for license.xml files. Licensing nodes will be stored in XML structures so license parts can easily be added or removed even without any license editor. The functions will be briefly discussed:

load file

Loads the license file selected in the path textbox.

save file

Saves the currently edited file to the path selected in the path textbox.

add node

Adds a license node to the currently selected node as parent node.

remove node

Removes the currently selected node. Please note that all subnodes will be removed as well.

create stub

If a license file is to be created from scratch, create stub can be used to create the root node.

test

Test will recursively test all nodes for their valididty on the current system. Can be used for license problem diagnosis on the target system. Once a license problem is detected the test will end, and the problem will be reported.

Node editor

Once a node is selected the node editor will automatically pop up. There you are able to edit all aspects of the current license node.

Name

The name of the node. This is the reference that will be browsed for if the license tree is checked.

Type

Type of the license node. Indicates what part of the system the license node is attached to. There are four different modes:

GLOBAL: The license node is not restricted and will be valid on any system.

CPU: The license node is restricted to the CPU serial number provided. If the system has more than once CPU core, the matching CPU will be looked for. If no installed CPU matches the code, the license will be invalid.

MAINBOARD: On a lot of systems the mainboard serial number is reported incorrectly. On some there is valid data available so a node can be theoretically connected to a mainboard.

DRIVE: The node is attached to a drive's serial number. All logical drives reported by the system can be used. If no drive containing the selected serial number is found, the node is invalid.

License Key

The key the node is attached to. Usually this is a serial number or something similar. The „fetch“ button can be used to retrieve all matching serial numbers for the currently selected type on the current system. Usually however this key will have to be provided manually since the license manager does not currently run on the target system. The customer can use the LicenseHelper application to provide the information to be used for licensing.

<- Fetch

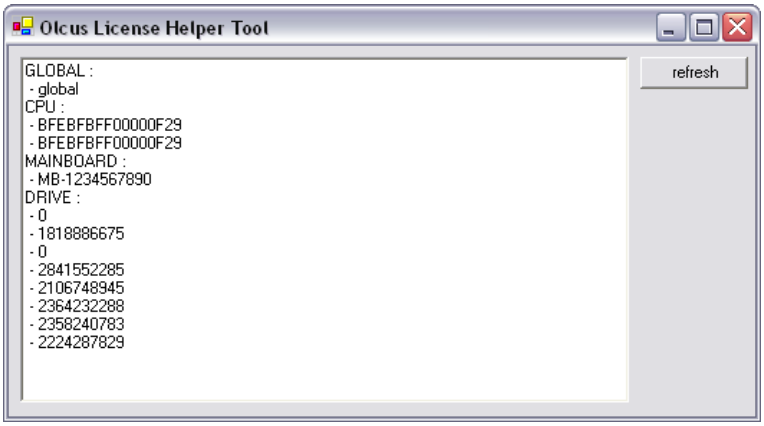
The fetch button will collect all serial numbers available for the currently selected license node type. The drop down menu of the License Key textfield will be filled with all serial numbers found. Once serial number will have to be selected. It is not possible to attach a node to more than one serial number. Two nodes would have to be used for that.

Expiration Date

License node can be restricted by an expiration date. If this function is not activated, there will be no limit how long the node is valid. If the checkbox is checked, the date selected will be the expiration date for the node. (keep in mind that the whole branch of the tree will become invalid once the expiration date is reached)

License Helper

The license application enables the customer to provide the information needed for creating a license file, without access to the license manager.



When started on the target system the license helper tool will automatically collect all serial numbers for all node types available. This infomration can then be cut and pasted by the customer to be send via email or whatever mean available.

Checking License information inside of Function Plugins

Access to the licensing system inside of a Plugin is provided by the LicenseServer object. The LicenseServer provides a static get method, that provides easy access to the only LicenseServer instance allowed per session. The LicenseServer instance provides the method checkLicensePath which can be used to make sure a certain path inside of the license tree is valid.

For example:

```
/// <summary>
/// This is mostly implemented for demonstration purposes
/// </summary>
/// <returns></returns>
public override bool checkLicense()
{
        // This is as simple as it gets. The license path is checked. If the license is not valid,
        // an exception will be thrown, and the parser execution stops here.
        return LicenseServer.get().checkLicensePath("core.plugins.foo");
}
```

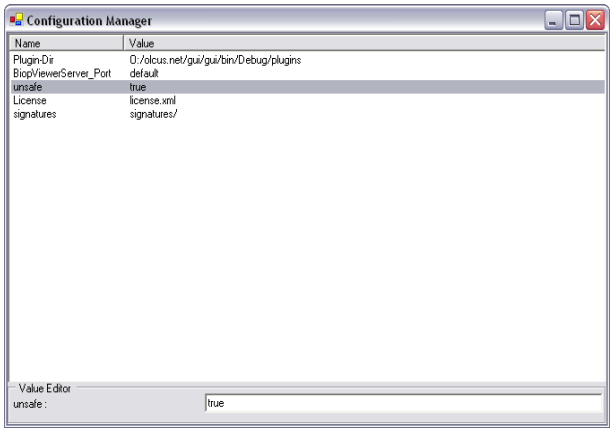
this will check the license path provided. If any of the indicated nodes is invalid, the whole license check will fail. An exception will be raised, with details about what node failed for what reason, so support should be able to solve any problems quickly.

The checkLicensePath method can be invoked from everywhere inside of your project. In this case the logic of the parser makes sure that execution will not continue if the license is invalid. In other cases the programmer might have to make sure that certain functionality is not accessable, when a license node is not valid.

Signatures

The Signature system is used to ensure that only supported plugins will be used with the olcus.net engine. While the olcus engine is designed to be robust, unpredictable side effects may occur from poorly implemented plugins. That would make supporting a script very difficult. The signature system allows to make sure that a support request is founded only on plugins that are trusted.

Safe/Unsafe mode

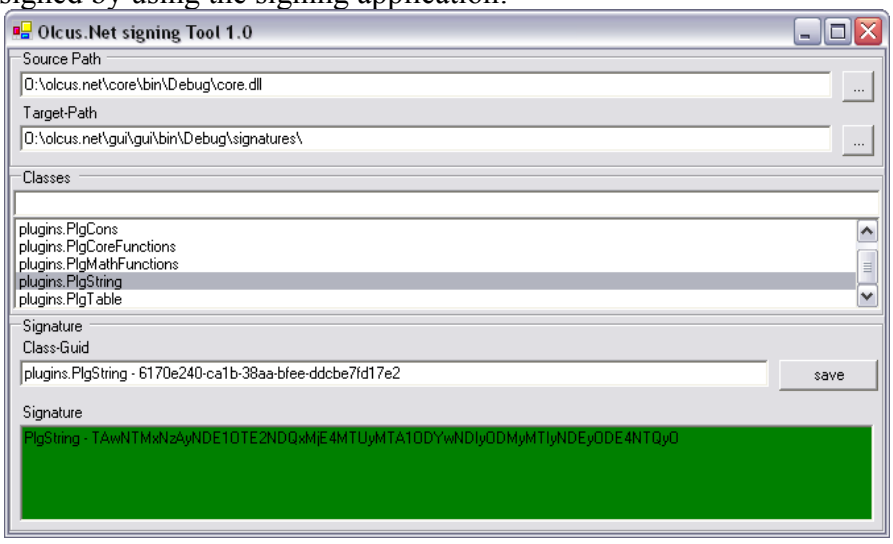


When olcus runs in safe mode, only trusted plugins are allowed to be used. If an unsafe plugin is referenced the parser will raise an exception and stop interpretation. Problems occuring during safe mode are valid for support.
During plugin development olcus can be switched into unsafe mode. All plugins will be accepted (unless their license is not valid). There will be no support for sessions running in unsafe mode.

The unsave mode is activated, when the parameters „unsafe“ is set to „true“ in the config file. You can easily do that inside of olcus' config manager. If unsafe is set to anything but „true“ the safe mode will be active.

Signing of Plugins

Plugins can be signed by using the signing application:



Source Path has to point to the DLL containing the plugin. Once a valid plugin DLL is selected, all plugins valid for signing are listed.
The target path should point to olcus „signatures“ directory. The directory that the olcus engine will browse looking for signature files. (That path can be changed in the config file)

Once „save“ is selected a signature file will be created. That signature file is valid for the selected plugin and version. Each .net type gets assigned an UUID on each compilation. So if the plugin is recompiled, it is neccessary to create a new signature. That is neccessary, so new versions of plugins can be reviewed before they, too, can be considered trusted.

Signature files can then be provided to the customer, that can deploy their plugins along with the corrisponding signature files.

Decoupler

Accessing gui objects inside of multithreaded system can be problematic. A gui component may only be accessed by the thread that created the gui enviroment. If a non gui-thread accesses a gui component the results are unpredictable. Usually that kind of problem manifests itself at the worst of moments (i.E. During a presentation).

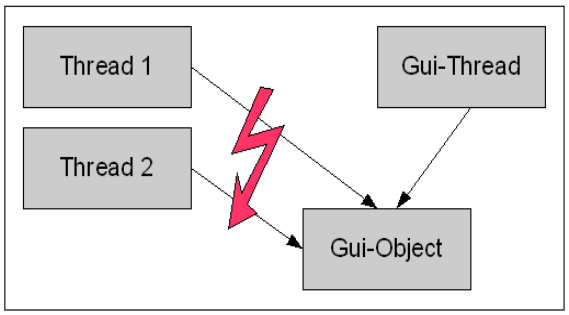


Fig 1: Threads my not call gui-objects directly.

The .net framework, which olcus.net is based on, allows to bypass such problems elegantly by the component.invoke method. That method basically passes a message to the gui thread, describing what method is to be called. The actual method call will then be done by the gui-thread, so everything will work normally. While this is a step in the right direction, it might still make multi-threaded gui code hard to implement, and complicated to debug. To make things easy to handle olcus.net implements a service called „Decoupler“.

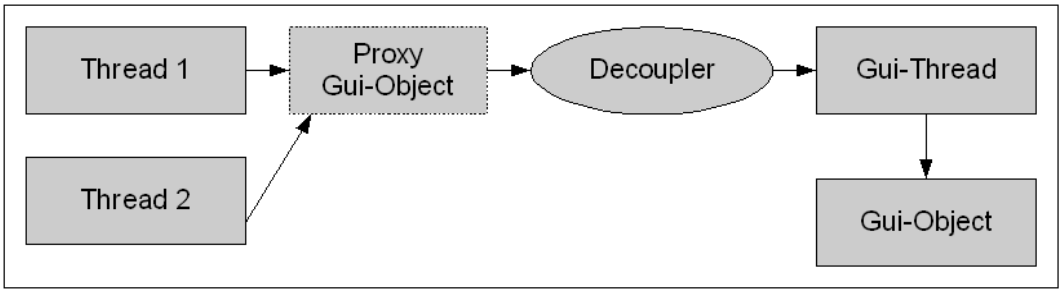


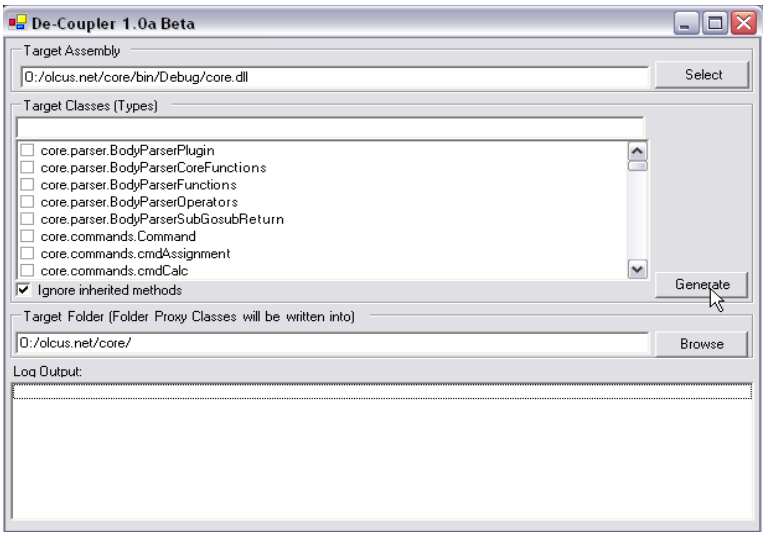
Fig 2: Decoupler syncronises calls, and decouples them from the actual thread

The Decoupler service allows the non-gui-thread to instance a proxy object, which will appear and behave just like the original gui-object, but will forward all method calls to the Decoupling service

that will then invoke the actual method inside of the original Gui-Thread.
That service two way-syncronous, so answers to the method call will be provided, as soon as the method call returns from execution.

Creating Proxy Objects

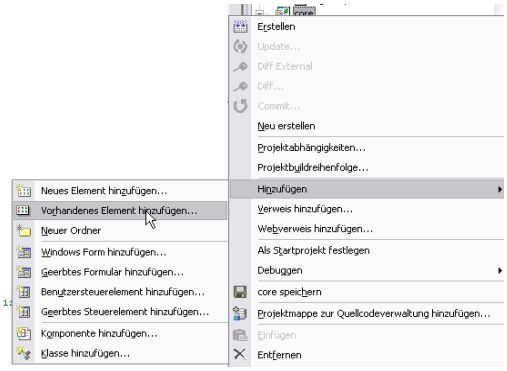
The proxy objects that are used and instanced by the non-gui thread, need to be created. To keep things as simple as possible, that is a fully automated process. The Decoupler Application is used to create these Proxy objects:



First you need to select the Plugin DLL containing your Gui-Classfiles. If your plugin is currently in development (which is most likely the case) please keep in mind that you need at least one successful compile of your class, for the de-coupler to be able to find it inside of the dll.
The Application will then display all classes available for the decoupling process. You can then select the classes you want to create Proxy Classes of.
The target folder is usually the current project file you are working on, of course you are free to generate the classfiles to whatever folder you want.
You can then select „Generate“ to create your proxy classes. They will be named after the classes they originate from, with the prefix „Iprx“. Should a proxy class already exist, it will be overwritten.

Please keep in mind that your proxy objects might become really large and slow if you uncheck „Ignore inherited methods“. It might be neccessary to access inherited methods in a lot of scenarios. However, to keep moderate filesizes, it should be a default setting to keep this option checked.

It might then be neccessary to include the new claesses to the project inside of Visual studio.



Using Proxy Objects

A proxy objects can be instanced like every other object. It will automatically set up the connection to the decoupler manager, which will instance itself, should it not already be running. All that has to be done is to create the object, and tell the object what real gui object its to be proxy for.

```
IprxMyGuiObject mgo = new IprxMyGuiObject(new MyGuiObject);
```

this will usually be part of the .create() method inside of a function plugin. Then you can access all methods that MyGuiObject posesses via the proxy object, while still maintaining a valid gui enviroment.

Static Methods for object creation

If you want to create the gui-object at runtime, from a non-gui thread you might find this workaround useful:

Create a static method which instances the object and returns it:

```
public static MyGuiObject create() {
  return new MyGuiObject();
}
```

Then when you instance the Proxy object you can use this static method to point to the object to be a proxy of:

```
IprxMyGuiObject mgo = new IpryMyGuiObject(MyGuiObject.create());
```

That way the actual instancing will be done by the gui-thread, and the proxy object can be used like any other decoupling proxy object.

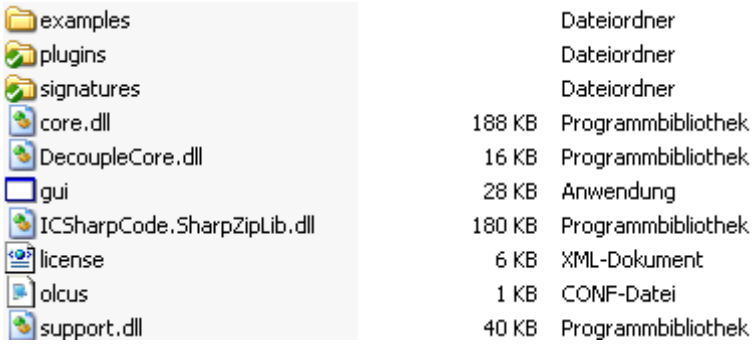
You can still use this method, even if your are planning on instancing the proxy class inside of the gui-thread, it will make little difference.

Deployment

There are two modes of Deployment for Olcus.net, End-Customer, and Plugin Developer.

End Customer Deployment:

The olcus application should be installed into a folder like that:



The number of DLL's may vary as olcus development progresses. It will usually suffice to copy the whole set of .dll files contained in the bin folder of the gui project.

Then the folder „examples“ should contain all example scripts, to ease development with the olcus engine.

The folder plugin can be empty at deployment, but might contain additionally purchased plugins.

The signatures folder is usually empty at deployment, too, since it will conly contain signatures for 3rd party plugins.

Both folders can basically be placed whereever the user wishes, its just important to set the folder position in the „olcus.conf“ file.

The license file can be placed in program root, or anywhere you like. Just make sure the conf file points to the correct place.

A folder „Documentation“ might be added containing the parts „Executor Manager“ and „Code Development“ of the olcus documentation.

The program is started via gui.exe, an icon should be placed on desktop.

LicenseHelper

During, or before installation the LicenseHelper.exe may be supplied to the customer, so he can report licensing information that can be used to create a licsense file.

Plugin Developer Deployment:

Plugin developers require some additional files, added to the normal user installation.

In addition to the files mentioned above the Plugin Developer ist supplied with:

- The additional documentation part „Plugin Documentation“
- The SDK, which will consist of an example plugin project, and maybe a plugin project template, if I find out how that is done.
- The Decoupler application

Olcus.Net 3.2 build 3578 Plugin reference file for plug-ins

Attention! This is a generated list of several plug-ins, some of which are only optional with your OLCUS-delivery or might require an additional fee. If you are missing a particular function or plug-in, please get in contact with FBI Science customer support; we can issue a quotation for you.

Plugin PlgArray Version: (3.2 build 3578)

The Array plugin can be used to store values indexed by other values. The behaviour is identical to that of a HashMap that is available in most object oriented systems.

Methods

put(index:String value:String)
Summary: Adds an indexed value to the arraylist
Parameter 'index': The index, used to retrieve the variable by later
Parameter 'value': The value we want to be stored
returns:

exists(index:String)
Summary: Returns true if a key exists, false, if it does not
Parameter 'index': The index key to be checked
returns:

pull(index:String)
Summary: Returns a value based on the index provided. If the index does not exist an error is thrown
Parameter 'index': The index the variable was previously stored under
returns:

Plugin PlgSerialPortSim Version: (3.2 build 3578)

This plugin simulates a serial port, for debugging of serial communication. A window is opened that displays all the data written into the serial port, and allows the simulation of data responses.

Methods

setBaud(baud:Integer)
Summary: Sets the baudrate for the connection
Parameter 'baud': The baudrate i.e. 9600
returns:

setStopBits(sb:Float)
Summary: Sets the number of stopbits. Possible values are 0, 1, 1.5, 2.
Parameter 'sb': The number of stopbits. Float.
returns:

setDataBits(db:Integer)
Summary: Sets the number of Databits per byte
Parameter 'db': The number of databits, integer
returns:

setParity(par:String)
Summary: Sets the parity for the SerialPort. Possible Values are even, mark, none, odd, space
Parameter 'par':
returns:

setHandshake(hs:String)
Summary: Sets the com port's handshaking. possible values are none, rts, both (xonxoff and rts), xonxoff
Parameter 'hs':
returns:

open()
Summary: Opens the connection to the com port
returns:

close()
Summary: Closes the Connection to the com component
returns:

setPort(port:String)
Summary: Sets the port to connect to.
Parameter 'port': The port i.E. com1:
returns:

setNewLine(newLine:String)
Summary: Sets the markers for a new line. Usually this will be \r for unix like clients, and \r\n for windows clients. Note that setNewLineUnix and setNewLineWindows can be used.
Parameter 'newLine': Any string that marks the beginning of a new line
returns:

setNewLineUnix()
Summary: Sets the marker for a new line to unix-friendly \r only.
returns:

setNewLineWindows()
Summary: Sets the marker for a new line to Windows(tm) friendly \r\n.
returns:

setReadTimeOut(timeout:Integer)
Summary: Sets the Read Timeout for Serial operations
Parameter 'timeout': The timeout for the reading operation in seconds

writeLine(str:String)

Summary: Sends a string to the Com Component, followed by carriage return linefeed
Parameter 'str': The String to be send
returns:

write(str:String)
Summary: Sends a string to the Com Component, no carriage return linefeed
Parameter 'str': The String to be send
returns:

writeByte(by:Integer)
Summary: Writes a single byte into the comport
Parameter 'by': The byte to be written
returns:

writeWord(word:Integer)
Summary: Writes a single word into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The word to be written
returns:

writeLong(by:Integer)
Summary: Writes a single longword into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The long to be written
returns:

readLineAsync()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readline()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readByteAsync()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readByte()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readWordAsync()
Summary: Reads a single word from the serial interface
returns:

readWord()
Summary: Reads a single word from the serial interface
returns:

readLongAsync()
Summary: Reads a single longword from the serial interface
returns:

readLong()
Summary: Reads a single longword from the serial interface
returns:

flush()
Summary: Makes the serial device flush the buffer, use only when autoflush is disabled.
returns:

setAutoFlush(auto:Boolean)
Summary: Enables or disables the autoflush feature. True will make the serial connection automatically flush all written Data, False will require the manual flushing via the .flush() method.
Parameter 'auto': True or False
returns:

Events

getdata
*description:*fires when new Data is available, provides data as a String.
*parameters:*OString
getRaw
*description:*fires when new Data is available, provides data as a Byte array.
*parameters:*OArray
getLine
*description:*fires when a line of text is available, the line end marker is determined by .setNewLine*. Line is Provided as String.
*parameters:*OString

Plugin PlgSerialPort Version: (3.2 build 3578)

This plugin simulates a serial port, for debugging of serial communication. A window is opened that displays all the data written into the serial port, and allows the simulation of data responses.

Methods

setBaud(baud:Integer)
Summary: Sets the baudrate for the connection
Parameter 'baud': The baudrate i.e. 9600
returns:

setStopBits(sb:Float)
Summary: Sets the number of stopbits. Possible values are 0, 1, 1.5, 2.

Parameter 'sb': The number of stopbits. Float.
returns:

setDataBits(db:Integer)
Summary: Sets the number of Databits per byte
Parameter 'db': The number of databits, integer
returns:

setParity(par:String)
Summary: Sets the parity for the SerialPort. Possible Values are even, mark, none, odd, space
Parameter 'par':
returns:

setHandshake(hs:String)
Summary: Sets the com port's handshaking. possible values are none, rts, both (xonxoff and rts), xonxoff
Parameter 'hs':
returns:

open()
Summary: Opens the connection to the com port
returns:

close()
Summary: Closes the Connection to the com component
returns:

setPort(port:String)
Summary: Sets the port to connect to.
Parameter 'port': The port i.E. com1:
returns:

setNewLine(newLine:String)
Summary: Sets the markers for a new line. Usually this will be \r for unix like clients, and \r\n for windows clients. Note that setNewLineUnix and setNewLineWindows can be used.
Parameter 'newLine': Any string that marks the beginning of a new line
returns:

setNewLineUnix()
Summary: Sets the marker for a new line to unix-friendly \r only.
returns:

setNewLineWindows()
Summary: Sets the marker for a new line to Windows(tm) friendly \r\n.
returns:

setReadTimeOut(timeout:Integer)
Summary: Sets the Read Timeout for Serial operations

Parameter 'timeout': The timeout for the reading operation in seconds

writeLine(str:String)
Summary: Sends a string to the Com Component, followed by carriage return linefeed
Parameter 'str': The String to be send
returns:

write(str:String)
Summary: Sends a string to the Com Component, no carriage return linefeed
Parameter 'str': The String to be send
returns:

writeByte(by:Integer)
Summary: Writes a single byte into the comport
Parameter 'by': The byte to be written
returns:

writeWord(word:Integer)
Summary: Writes a single word into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The word to be written
returns:

writeLong(by:Integer)
Summary: Writes a single longword into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The long to be written
returns:

readLineAsync()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readline()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readByteAsync()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readByte()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readWordAsync()

Summary: Reads a single word from the serial interface
returns:

readWord()
Summary: Reads a single word from the serial interface
returns:

readLongAsync()
Summary: Reads a single longword from the serial interface
returns:

readLong()
Summary: Reads a single longword from the serial interface
returns:

flush()
Summary: Makes the serial device flush the buffer, use only when autoflush is disabled.
returns:

setAutoFlush(auto:Boolean)
Summary: Enables or disables the autoflush feature. True will make the serial connection automatically flush all written Data, False will require the manual flushing via the .flush() method.
Parameter 'auto': True or False
returns:

Events

getdata
*description:*fires when new Data is available, provides data as a String.
*parameters:*OString
getRaw
*description:*fires when new Data is available, provides data as a Byte array.
*parameters:*OArray
getLine
*description:*fires when a line of text is available, the line end marker is determined by .setNewLine*. Line is Provided as String.
*parameters:*OString

Plugin PlgSocketSim Version: (3.2 build 3578)

This class can be used to talk to any TCP/IP socket based service

Methods

open(addr:String port:Integer)
Summary: Opens a socket connection to the specified port and tcp / ip adress
Parameter 'addr':
Parameter 'port':
returns:

open(port:Integer)
Summary: Opens a socket connection to the specified port and tcp / ip adress
Parameter 'addr':
Parameter 'port':
returns:

close()
Summary: Closes the socket connection
returns:

open()
Summary: Opens the socket with previously supplied address and port
returns:

setAddress(address:String)
Summary: Sets the IPAdress or hostname to be used by the common socket
Parameter 'address': The adress to be used
returns:

setPort(port:Integer)
Summary: Sets the port to be used by a later open command
Parameter 'port':
returns:

open(addr:String port:Integer)
Summary: Opens the socket with supplied adress and port
Parameter 'addr':
Parameter 'port':
returns:

setNewLine(newLine:String)
Summary: Sets the markers for a new line. Usually this will be \r for unix like clients, and \r\n for windows clients. Note that setNewLineUnix and setNewLineWindows can be used.
Parameter 'newLine': Any string that marks the beginning of a new line
returns:

setNewLineUnix()
Summary: Sets the marker for a new line to unix-friendly \r only.
returns:

setNewLineWindows()
Summary: Sets the marker for a new line to Windows(tm) friendly \r\n.
returns:

setReadTimeOut(timeout:Integer)
Summary: Sets the Read Timeout for Serial operations
Parameter 'timeout': The timeout for the reading operation in seconds

writeLine(str:String)
Summary: Sends a string to the Com Component, followed by carriage return linefeed
Parameter 'str': The String to be send
returns:

write(str:String)
Summary: Sends a string to the Com Component, no carriage return linefeed
Parameter 'str': The String to be send
returns:

writeByte(by:Integer)
Summary: Writes a single byte into the comport
Parameter 'by': The byte to be written
returns:

writeWord(word:Integer)
Summary: Writes a single word into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The word to be written
returns:

writeLong(by:Integer)
Summary: Writes a single longword into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The long to be written
returns:

readLineAsync()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readline()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readByteAsync()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readByte()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readWordAsync()
Summary: Reads a single word from the serial interface

returns:

readWord()
Summary: Reads a single word from the serial interface
returns:

readLongAsync()
Summary: Reads a single longword from the serial interface
returns:

readLong()
Summary: Reads a single longword from the serial interface
returns:

flush()
Summary: Makes the serial device flush the buffer, use only when autoflush is disabled.
returns:

setAutoFlush(auto:Boolean)
Summary: Enables or disables the autoflush feature. True will make the serial connection automatically flush all written Data, False will require the manual flushing via the .flush() method.
Parameter 'auto': True or False
returns:

Events

getdata
*description:*fires when new Data is available, provides data as a String.
*parameters:*OString
getRaw
*description:*fires when new Data is available, provides data as a Byte array.
*parameters:*OArray
getLine
*description:*fires when a line of text is available, the line end marker is determined by .setNewLine*. Line is Provided as String.
*parameters:*OString

Plugin PlgSocketClient Version: (3.2 build 3578)

This plugin provides functions to communicate with any TCP/IP socket server.

Methods

open(addr:String port:Integer)
Summary: Opens a socket connection to the specified port and tcp / ip adress
Parameter 'addr':
Parameter 'port':
returns:

close()
Summary: Closes the socket connection
returns:

open()
Summary: Opens the socket with previously supplied address and port
returns:

setAddress(address:String)
Summary: Sets the IPAdress or hostname to be used by the common socket
Parameter 'address': The adress to be used
returns:

setPort(port:Integer)
Summary: Sets the port to be used by a later open command
Parameter 'port':
returns:

open(addr:String port:Integer)
Summary: Opens the socket with supplied adress and port
Parameter 'addr':
Parameter 'port':
returns:

setNewLine(newLine:String)
Summary: Sets the markers for a new line. Usually this will be \r for unix like clients, and \r\n for windows clients. Note that setNewLineUnix and setNewLineWindows can be used.
Parameter 'newLine': Any string that marks the beginning of a new line
returns:

setNewLineUnix()
Summary: Sets the marker for a new line to unix-friendly \r only.
returns:

setNewLineWindows()
Summary: Sets the marker for a new line to Windows(tm) friendly \r\n.
returns:

setReadTimeOut(timeout:Integer)
Summary: Sets the Read Timeout for Serial operations
Parameter 'timeout': The timeout for the reading operation in seconds

writeLine(str:String)
Summary: Sends a string to the Com Component, followed by carriage return linefeed
Parameter 'str': The String to be send
returns:

write(str:String)

Summary: Sends a string to the Com Component, no carriage return linefeed
Parameter 'str': The String to be send
returns:

writeByte(by:Integer)
Summary: Writes a single byte into the comport
Parameter 'by': The byte to be written
returns:

writeWord(word:Integer)
Summary: Writes a single word into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The word to be written
returns:

writeLong(by:Integer)
Summary: Writes a single longword into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The long to be written
returns:

readLineAsync()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readline()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readByteAsync()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readByte()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readWordAsync()
Summary: Reads a single word from the serial interface
returns:

readWord()
Summary: Reads a single word from the serial interface
returns:

readLongAsync()

Summary: Reads a single longword from the serial interface
returns:

readLong()
Summary: Reads a single longword from the serial interface
returns:

flush()
Summary: Makes the serial device flush the buffer, use only when autoflush is disabled.
returns:

setAutoFlush(auto:Boolean)
Summary: Enables or disables the autoflush feature. True will make the serial connection automatically flush all written Data, False will require the manual flushing via the .flush() method.
Parameter 'auto': True or False
returns:

Events

getdata
*description:*fires when new Data is available, provides data as a String.
*parameters:*OString
getRaw
*description:*fires when new Data is available, provides data as a Byte array.
*parameters:*OArray
getLine
*description:*fires when a line of text is available, the line end marker is determined by .setNewLine*. Line is Provided as String.
*parameters:*OString

Plugin PlgMathFunctions Version: (3.2 build 3578)

The Math plugin provides access to the most common mathematical functions. The Math plugin does not need to be instanced. Its automatically available by the name "math" in every olcus script.

Methods

random(range:Integer)
Summary: Returns a random number inside of the specified range (0-range)
Parameter 'range': The upper end of the range, starting with the number 0
returns: A random number as integer

maxInt()
Summary: Returns the maximum integer value
returns: The maximum

minInt()
Summary: Returns the minimum integer value
returns: The minimum

abs(value:Integer)
Summary: Returns the absolute value of the passed value. Returned value is always positive
Parameter 'value': The value, positive or negative
returns:

Acos(value:Double)
Summary: Returns the angle whose cosine is the specified number.
Parameter 'value': cosine
returns:

ssin(value:Double)
Summary: Returns the angle whose sine is the specified number.
Parameter 'value': The sine
returns:

atan(value:Double)
Summary: Returns the angle whose tangent is the specified number.
Parameter 'value': The angle
returns:

atan2(x:Double y:Double)
Summary: Returns the angle whose tangent is the quotient of two specified numbers.
Parameter 'value':
returns:

bigMul(a:Integer b:Integer)
Summary: Produces the full product of two 32-bit numbers.
Parameter 'a': Integer a
Parameter 'b': Integer b
returns:

ceiling(value:Double)
Summary: Returns the smallest integer greater than or equal to the specified double-precision floating-point number.
Parameter 'value': The float
returns:

cos(value:Double)
Summary: Returns the cosine of the specified angle.
Parameter 'value':
returns:

cosh(value:Double)
Summary: Returns the hyperbolic cosine of the specified angle.
Parameter 'value':
returns:

divRem(a:Integer b:Integer)
Summary: Calculates the quotient of two integers and returns the remainder
Parameter 'a': Value a
Parameter 'b': Value b
returns:

exp(value:Double)
Summary: Returns e raised to the specified power.
Parameter 'value': The value
returns:

floor(value:Double)
Summary: Returns the largest integer less than or equal to the specified double-precision floating-point number.
Parameter 'value': The floating point number
returns:

IEEERemainder(x:Double y:Double)
Summary: Returns the remainder resulting from the division of a specified number by another specified number.
Parameter 'x': Value X
Parameter 'y': Value Y
returns:

log(value:Double)
Summary: Returns the natural (base e) logarithm of a specified number.
Parameter 'value': The number
returns:

log(a:Double b:Double)
Summary: Returns the logarithm of a specified number in a specified base.
Parameter 'a': the number
Parameter 'b': the base
returns:

log10(value:Double)
Summary: Returns the base 10 logarithm of a specified number.
Parameter 'value': The number
returns:

max(a:Double b:Double)
Summary: Returns the larger of two specified numbers.
Parameter 'a': number a
Parameter 'b': number b
returns:

max(a:Float b:Float)
Summary: Returns the larger of two specified numbers.

Parameter 'a': number a
Parameter 'b': number b
returns:

max(a:Integer b:Integer)
Summary: Returns the larger of two specified numbers.
Parameter 'a': number a
Parameter 'b': number b
returns:

max(a:Long b:Long)
Summary: Returns the larger of two specified numbers.
Parameter 'a': number a
Parameter 'b': number b
returns:

min(a:Double b:Double)
Summary: Returns the smaller of two numbers.
Parameter 'a': number a
Parameter 'b': number b
returns:

min(a:Float b:Float)
Summary: Returns the smaller of two numbers.
Parameter 'a': number a
Parameter 'b': number b
returns:

min(a:Long b:Long)
Summary: Returns the smaller of two numbers.
Parameter 'a': number a
Parameter 'b': number b
returns:

min(a:Integer b:Integer)
Summary: Returns the smaller of two numbers.
Parameter 'a': number a
Parameter 'b': number b
returns:

pow(a:Double b:Double)
Summary: Returns a specified number raised to the specified power.
Parameter 'a': number
Parameter 'b': the power
returns:

round(toRound:Double numOfDecimals:Integer)
Summary: Rounds a value to the specified number of decimal places.

Parameter 'toRound': The double to be rounded
Parameter 'numOfDecimals': The number of decimals to round to
returns:

round(toRound:Double)
Summary: Rounds a double Value into an Integer (no decimals)
Parameter 'toRound': The double to round
returns:

sign(value:Double)
Summary: Returns a value indicating the sign of a double number.
Parameter 'value': the number
returns:

sign(value:Integer)
Summary: Returns a value indicating the sign of an integer.
Parameter 'value':
returns:

sin(value:Double)
Summary: Returns the sine of the specified angle.
Parameter 'value': The angle
returns:

sinh(value:Double)
Summary: Returns the hyperbolic sine of the specified angle.
Parameter 'value': The angle
returns:

sqrt(value:Double)
Summary: Returns the square root of a specified number.
Parameter 'value':
returns:

tan(value:Double)
Summary: Returns the tangent of the specified angle.
Parameter 'value': The angle
returns:

tanh(value:Double)
Summary: Returns the hyperbolic tangent of the specified angle.
Parameter 'value': The angle
returns:

truncate(value:Double)
Summary: Calculates the integral part of a specified double number.
Parameter 'value': the double
returns:

Plugin PlgString Version: (3.2 build 3578)

The string plugin provides functions needed for string operations The String plugin does not need to be instanced. Its automatically available under the name "String" for every olcus script.

Methods

sub(str:String start:Integer end:Integer)
Documentation unavailable
subByLen(str:String start:Integer len:Integer)
Summary: Returns a sub part of the selected String
Parameter 'str': The string to cut
Parameter 'start': Start index (>=0)
Parameter 'len': String length
returns:

left(str:String len:Integer)
Summary: Returns the left part of a string with the specified length
Parameter 'str': The string to cut
Parameter 'len': The length to cut
returns: The part of the string to return

right(str:String len:Integer)
Summary: Returns the right part of a string with the specified length
Parameter 'str': The string to cut
Parameter 'len': The length to cut
returns: The part of the string to return

indexOf(str:String pattern:String)
Summary: Returns the index of a pattern inside of the string, or -1 if the pattern is not found
Parameter 'str': The string to search in
Parameter 'pattern': The pattern to search for
returns: Index of the pattern

length(str:String)
Summary: Returns the lenght of a String in characters
Parameter 'str': The string to measure
returns:

len(str:String)
Summary: Returns the length of a String in characters, Identical to length, just for the lazy
Parameter 'str': The string to measure
returns:

intToHex(num:Integer)
Summary: Converts the submitted number into a hexadecimal representation of said number
Parameter 'num': The number

returns:

arrayToHex(arr:Array)
Summary: Converts integer values contained inside of an array into a hexadecimal representation
Parameter 'arr': The array to convert
returns: A string in the form of 00010203...

arrayToHex(arr:Array seperator:String)
Summary: Converts integer values contained inside of an array into a hexadecimal representation
Parameter 'arr': The array to convert
Parameter 'seperator': The char to seperate the array by, for example ";"
returns: A string in the form of 00;01;02;03...

split(toSplit:String seperator:String)
Summary: Splits a string according to the provided seperator. The method returns an array of strings. So new (a:array) = String.split("a b c", " ") will create an array containing the strings "a", "b" and "c"
Parameter 'toSplit': The string to be split
Parameter 'seperator': The seperator
returns:

Plugin PlgTurboTable Version: (3.2 build 3578)

An alternative Table implementation. Can be used when table updates occur very frequently. The regular table model is a bit more flexible, while this performs much better.

Methods

setTitle(str:String)
Summary: Sets the title of the table window
Parameter 'str': The title as String
returns:

addColumn(str:String)
Summary: Adds a Column to the table.
Parameter 'str': Title of the Column
returns: Nothing

addRow()
Summary: Context variable can be used if things need to be done inside of the local threads context. The parser will automatically assign the context object, so that the script only needs to supply the other variables.
Parameter 'cont': Context in which this function is called
returns: Nothing. In this context this row will be locked, though

setValue(column:String val:String)
Summary: Sets a value in current row, in the column indexed by its name
Parameter 'cont': Context is provided automatically by function call plugin system

Parameter 'column': The column to use
Parameter 'val': The value to set
returns:

setValue(column:Integer val:String)
Summary: Sets a value in current row, in the column indexed by a number
Parameter 'cont': Context is provided automatically by function call plugin system
Parameter 'column': The column index, first column is 1, second is 2, and so on.
Parameter 'val': The value to set
returns:

setRow(row:Integer)
Summary: Sets the row to be used in the table. Be advised that serious threading problems may occur if this is used in the wrong way. The same row may never be manipulated by two threads at once.
Parameter 'cont': Procided by the function call
Parameter 'row': An integer index to the row to be used, starting with 0
returns:

setColSize(column:Integer size:Integer)
Summary: Sets the size of a column in a table
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

setColAlign(column:Integer align:Integer)
Summary: Sets the size of a column in a table
Parameter 'column': The column to set the alignement for
Parameter 'align': The alignment to be set: 1 - left 2 - right 3 - center
returns:

setColAlign(column:String align:Integer)
Summary: Sets the size of a column in a table
Parameter 'column': The column to set the alignement for
Parameter 'align': The alignment to be set: 1 - left 2 - right 3 - center
returns:

setColSize(column:String size:Integer)
Summary: Sets the size of a column in a table
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

autoSizeCol(column:Integer)
Summary: Automatically adjusts the size of a column
Parameter 'cont': Automatically provided

Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

autoSizeCol(column:String)
Summary: Automatically adjusts the size of a column
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

autoSize()
Summary: Automatically adjusts the size of a column
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

setDockMode(mode:Boolean)
Summary: Determines if we want the window to be created in "docked" mode. If this is set to false the window will be a normal window, otherwise it will be created a document docked on creation.
Parameter 'mode': True will allow the window to dock, false will disallow it
returns: Nothing.

saveToFile(fileName:String)
Summary: Saves the whole content of the table into a specified file Threadsafe, because called methods are threadsafe.
Parameter 'fileName': The filename to save to
returns:

saveTableToFileWriter(fileWriter:Plugin)
Summary: Saves the whole content of the table into a fileWriter
Parameter 'fileWriter': The filewriter to save into
returns:

saveRowToFileWriter(fileWriter:Plugin)
Summary: Saves the currently active row into the fileWriter
Parameter 'fileWriter': The filewriter to use
returns:

activateAutosave(filename:String interval:Integer)
Summary: Activates the autosave funktion of the table, will save periodically into the specified file
Parameter 'filename': The filename to save to
Parameter 'interval': The interval between saves in miliseconds
returns: Does not return anything

deactivateAutosave()
Summary: Deactivates the autosaving feature in case its no longe wanted. If the autosave was active

this will also make the feature save one final dump of the table contents into the specified filename. This is also automatically called whenever the program stops execution
returns:

Events

windowClosed
*description:*Fires when the window is closed
*parameters:*unkown parameter types

Plugin PlgLogger Version: (3.2 build 3578)

This plugin is used to log messages into a file

Methods

open(path:String)
Summary: Opens a logfile for logging messages into
Parameter 'path':
returns:

setLevel(level:Integer)
Summary: Sets the logging level that controls which messages are discareded and which ones are used
Parameter 'level': The loglevel, refer to the olcus.net manual
returns:

log(level:Integer msg:String)
Summary: Writes a log message into the file
Parameter 'level': The level of the message
Parameter 'msg': The textmessage
returns:

log(msg:String)
Summary: Writes a log message into the file The info loglevel will be used by default
Parameter 'msg': The textmessage
returns:

info()
Summary: Returns the info loglevel to be used by the log method.
returns:

warning()
Summary: Returns the warning loglevel to be used by the log method.
returns:

error()
Summary: Returns the error loglevel to be used by the log method.
returns:

debug()
Summary: Returns the debug loglevel to be used by the log method.
returns:

critical()
Summary: Returns the critical loglevel to be used by the log method.

Plugin PlgEmail Version: (3.2)

No description available

Methods

initSMTP(smtpServer:String port:Integer useSSL:Boolean user:String pass:String)
Summary: Initializes the SMTP server to be used by the SendEmail command
Parameter 'smtpServer': The smtp server host name or IP adress
Parameter 'port': The port to be used
returns:

initSMTP(smtpServer:String port:Integer)
Summary: Initializes the SMTP server to be used by the SendEmail command
Parameter 'smtpServer': The Server to be used
returns:

initSMTP(smtpServer:String)
Summary: Initializes the SMTP server to be used by the SendEmail command
Parameter 'smtpServer': The Server to be used
returns:

setDefaultFromEmail(from:String)
Summary: Sets the Default adress emails originate from. Important when the SMTP server checks the "From" field. Due to spam prevention most email servers do that these days.
Parameter 'from': An email adress e.G. someone@some.net
returns:

sendEmail(from:String to:String subject:String body:String)
Summary: Allows sending an email via unencrypted SMTP servers Make sure the initSMTP command was used first.
Parameter 'from': The origin address e.G. someone@some.net
Parameter 'to': The target address e.G. someone@some.net
Parameter 'subject': The email subject
Parameter 'body': The email body
returns:

sendEmail(to:String subject:String body:String)
Summary: Allows sending an email via unencrypted SMTP servers Make sure the initSMTP command was used first.
Parameter 'to': The target address e.G. someone@some.net

Parameter 'subject': The email subject
Parameter 'body': The email body
returns:

sendEmail(to:String message:String)
Summary: Simplified version of the send email command.
Parameter 'to': The target address e.G. someone@some.net
Parameter 'message': The message (subject and body)
returns:

Plugin PlgFileWriter Version: (3.2 build 3578)

The FileWriter plugins allows to write data into a textfile. One Plugin needs to be instanced for every textfile to write data into.

Methods

createFile(fname:String overwrite:Boolean)
Summary: Creates a textfile to write into
Parameter 'fname': The filename of the file
Parameter 'overwrite': Set to true, if you want to allow files to be overwritten
returns:

createFile(fname:String)
Summary: Creates a textfile to write into
Parameter 'fname': The filename of the file
returns:

appendFile(fname:String)
Summary: Appends to an existing textfile. If the file does not exist, it is created.
Parameter 'fname': The filename to append to
returns:

setNewLineUnix()
Summary: Sets the stream to unix mode, meaning that only linefeed is used
returns:

setNewLineWindows()
Summary: Sets the stream to Windows mode, meaning that Carriage Return Linefeed is going to be used
returns:

write(data:String)
Summary: Writes data into the file without carriage return.
Parameter 'data': The data to be written
returns:

writeLine(data:String)

Summary: Writes a line of text into the file, ending with a carriage return Depending on the mode set (unix or windows) the corresponding line end is attached
Parameter 'data':
returns:

close()
Summary: Closes the file
returns:

Plugin PlgFileReader Version: (3.2 build 3578)

The FileReader plugin provides functions to read data from a text file. One FileReader plugin should be instanced for every file used.

Methods

eof()
Summary: Checks to see if the end of file is reached
returns: True if there is no more data to read, otherwise false

open(fname:String)
Summary: Opens a file
Parameter 'fname': The file's path
returns:

read()
Summary: Reads one char from the stream
returns:

readLine()
Summary: Reads a line of text from the file
returns:

close()
Summary: Closes the file
returns:

Plugin PlgNamedPipeClient Version: (3.2 build 3578)

This plugin allows to establish named pipe connections from the olcus application. This client can connect to an existing server and exchange data.

Methods

setName(pname:String)
Summary: Sets the name of the named pipe to connect to
Parameter 'pname': The pipename, E.g. \\.\pipe\myNamedPipe
returns:

open()
Summary: Opens the pipe, using the previously specified pipename
returns:

close()
Summary: Closes the pipe connection
returns:

open(pipename:String)
Summary: Opens the pipe, using the provided pipename
Parameter 'pipename': The pipename, E.g. \\.\pipe\myNamedPipe
returns:

setNewLine(newLine:String)
Summary: Sets the markers for a new line. Usually this will be \r for unix like clients, and \r\n for windows clients. Note that setNewLineUnix and setNewLineWindows can be used.
Parameter 'newLine': Any string that marks the beginning of a new line
returns:

setNewLineUnix()
Summary: Sets the marker for a new line to unix-friendly \r only.
returns:

setNewLineWindows()
Summary: Sets the marker for a new line to Windows(tm) friendly \r\n.
returns:

setReadTimeOut(timeout:Integer)
Summary: Sets the Read Timeout for Serial operations
Parameter 'timeout': The timeout for the reading operation in seconds

writeLine(str:String)
Summary: Sends a string to the Com Component, followed by carriage return linefeed
Parameter 'str': The String to be send
returns:

write(str:String)
Summary: Sends a string to the Com Component, no carriage return linefeed
Parameter 'str': The String to be send
returns:

writeByte(by:Integer)
Summary: Writes a single byte into the comport
Parameter 'by': The byte to be written
returns:

writeWord(word:Integer)

Summary: Writes a single word into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The word to be written
returns:

writeLong(by:Integer)
Summary: Writes a single longword into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The long to be written
returns:

readLineAsync()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readline()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readByteAsync()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readByte()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readWordAsync()
Summary: Reads a single word from the serial interface
returns:

readWord()
Summary: Reads a single word from the serial interface
returns:

readLongAsync()
Summary: Reads a single longword from the serial interface
returns:

readLong()
Summary: Reads a single longword from the serial interface
returns:

flush()
Summary: Makes the serial device flush the buffer, use only when autoflush is disabled.
returns:

setAutoFlush(auto:Boolean)
Summary: Enables or disables the autoflush feature. True will make the serial connection automatically flush all written Data, False will require the manual flushing via the .flush() method.
Parameter 'auto': True or False
returns:

Events

getdata
*description:*fires when new Data is available, provides data as a String.
*parameters:*OString
getRaw
*description:*fires when new Data is available, provides data as a Byte array.
*parameters:*OArray
getLine
*description:*fires when a line of text is available, the line end marker is determined by .setNewLine*. Line is Provided as String.
*parameters:*OString

Plugin PlgNamedPipeServer Version: (3.2 build 3578)

This plugin allows to create a named pipe server. Clients can then connect to the named pipe server and exchange data.

Methods

setName(pname:String)
Summary: Sets the name of the named pipe to connect to
Parameter 'pname': The pipename, E.g. \\.\pipe\myNamedPipe
returns:

open()
Summary: Opens the pipe, using the previously specified pipename
returns:

close()
Summary: Closes the pipe connection
returns:

open(pipename:String)
Summary: Opens the pipe, using the provided pipename
Parameter 'pipename': The pipename, E.g. \\.\pipe\myNamedPipe
returns:

setNewLine(newLine:String)
Summary: Sets the markers for a new line. Usually this will be \r for unix like clients, and \r\n for windows clients. Note that setNewLineUnix and setNewLineWindows can be used.
Parameter 'newLine': Any string that marks the beginning of a new line

returns:

setNewLineUnix()
Summary: Sets the marker for a new line to unix-friendly \r only.
returns:

setNewLineWindows()
Summary: Sets the marker for a new line to Windows(tm) friendly \r\n.
returns:

setReadTimeOut(timeout:Integer)
Summary: Sets the Read Timeout for Serial operations
Parameter 'timeout': The timeout for the reading operation in seconds

writeLine(str:String)
Summary: Sends a string to the Com Component, followed by carriage return linefeed
Parameter 'str': The String to be send
returns:

write(str:String)
Summary: Sends a string to the Com Component, no carriage return linefeed
Parameter 'str': The String to be send
returns:

writeByte(by:Integer)
Summary: Writes a single byte into the comport
Parameter 'by': The byte to be written
returns:

writeWord(word:Integer)
Summary: Writes a single word into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The word to be written
returns:

writeLong(by:Integer)
Summary: Writes a single longword into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The long to be written
returns:

readLineAsync()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readline()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readByteAsync()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readByte()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readWordAsync()
Summary: Reads a single word from the serial interface
returns:

readWord()
Summary: Reads a single word from the serial interface
returns:

readLongAsync()
Summary: Reads a single longword from the serial interface
returns:

readLong()
Summary: Reads a single longword from the serial interface
returns:

flush()
Summary: Makes the serial device flush the buffer, use only when autoflush is disabled.
returns:

setAutoFlush(auto:Boolean)
Summary: Enables or disables the autoflush feature. True will make the serial connection automatically flush all written Data, False will require the manual flushing via the .flush() method.
Parameter 'auto': True or False
returns:

Events

getdata
*description:*fires when new Data is available, provides data as a String.
*parameters:*OString
getRaw
*description:*fires when new Data is available, provides data as a Byte array.
*parameters:*OArray
getLine
*description:*fires when a line of text is available, the line end marker is determined by .setNewLine*. Line is Provided as String.

*parameters:*OString

Plugin PlgCoreFunctions Version: (3.2 build 3578)

The CoreFunctions plugin provides access to system functions of the host computer. This Plugin does not need to be instanced manually, it can be globally referenced by "system" for every olcus script.

Methods

sleep(interval:Integer)
Summary: Suspends the current executor for the specified amount of time
Parameter 'interval': The amount of miliseconds to sleep by
returns: Returns void

msgbox(str:String)
Summary: Opens a message box, to display the provided string
Parameter 'str': Message to be displayed
returns: Returns nothing

alert(str:String)
Summary: Opens a messagebox and displays the string. This method is deprecated, and implemented for backwards compatability.
Parameter 'str': String to be displayed
returns:

inputBox(message:String)
Summary: Opens a text input box, waits for user input, and then returns that input. While the box is active the current execution thread is paused.
Parameter 'cont': Needs context object so it can stop waiting when the program stops.
returns: A String witht he user input

inputBox()
Summary: Opens a text input box, waits for user input, and then returns that input. While the box is active the current execution thread is paused.
Parameter 'cont': Content object
returns:

militime()
Summary: Returns the number of Miliseconds since the System booted up
returns: number of miliseconds

dateTime()
Summary: Returns a timestamp as String
returns: The current timestamp

dateTime(fmt:String)
Summary: Returns the date time according to the provided format.
Parameter 'fmt': Example: dd.MM.yyyy hh:mm:ss

returns:

date()
Summary: Returns a timestamp as String
returns: The current timestamp

time()
Summary: Returns the current system Time in the form of hh:mm:ss
returns:

militimeToDateTime(militime:Integer fmt:String)
Summary: This function will convert a militime into a formatted time string
Parameter 'militime': The militime value
Parameter 'format': The format String, exmaple: HH:mm:ss
returns:

exit()
Summary: This methods ends the currently exectuting program completely. In comparison to the "end" command that will only end the current executor, this command will shutdown all running executors.
returns:

getEnvironment(env:String)
Summary: Returns the content of en environment variable.
Parameter 'env': The name of the Environment vairable e.g. "USERNAME"
returns:

getCommandLineArg(index:Integer)
Summary: Returns a parameter that was passed to the application on execution
Parameter 'index': The index of the passed parameter
returns:

netSend(targetHost:String message:String)
Summary: Sends a message to a target host via the net send command
Parameter 'targetHost': The Target host to send the message to.
Parameter 'message': The message to be send
returns:

Plugin PlgCons Version: (3.2 build 3578)

This plugin allows to output text to the Logging System. There is no need to create an instance of the console plugin, since its globally assigned to the name "console" for every olcus script.

Methods

echo(str:String)
Summary: Echos a string to the console, or the currently active logging system
Parameter 'str':

returns:

input()
Summary: Waits for a string from the console. Does not make sense unless olcus is actually run in console mode.
returns: Returns the returned stirng

Plugin PlgTable Version: (3.2 build 3578)

The table Plugin allows to display experiment related data in table form. The respective Table window also offers export functions for the displayed data. On Table plugin per diplayed table has to be instanced.

Methods

setTitle(str:String)
Summary: Sets the title of the table window
Parameter 'str': The title as String
returns:

setPos(x:Integer y:Integer)
Summary: Sets the position of the table, relative to upper left corner of the first screen
Parameter 'x': The x-position on the screen
Parameter 'y': The y-position on the screen

addColumn(str:String)
Summary: Adds a Column to the table.
Parameter 'str': Title of the Column
returns: Nothing

addRow()
Summary: Context variable can be used if things need to be done inside of the local threads context. The parser will automatically assign the context object, so that the script only needs to supply the other variables.
Parameter 'cont': Context in which this function is called
returns: Nothing. In this context this row will be locked, though

setValue(column:String val:String)
Summary: Sets a value in current row, in the column indexed by its name
Parameter 'cont': Context is provided automatically by function call plugin system
Parameter 'column': The column to use
Parameter 'val': The value to set
returns:

setValue(column:Integer val:String)
Summary: Sets a value in current row, in the column indexed by a number
Parameter 'cont': Context is provided automatically by function call plugin system
Parameter 'column': The column index, first column is 1, second is 2, and so on.

Parameter 'val': The value to set
returns:

setRow(row:Integer)
Summary: Sets the row to be used in the table. Be advised that serious threading problems may occur if this is used in the wrong way. The same row may never be manipulated by two threads at once.
Parameter 'cont': Procided by the function call
Parameter 'row': An integer index to the row to be used, starting with 0
returns:

setColSize(column:Integer size:Integer)
Summary: Sets the size of a column in a table
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

setColSize(column:String size:Integer)
Summary: Sets the size of a column in a table
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

autoSizeCol(column:Integer)
Summary: Automatically adjusts the size of a column
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

autoSizeCol(column:String)
Summary: Automatically adjusts the size of a column
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

autoSize()
Summary: Automatically adjusts the size of a column
Parameter 'cont': Automatically provided
Parameter 'column': The column to set the size for
Parameter 'size': The size to be set
returns:

setDockMode(mode:Boolean)
Summary: Determines if we want the window to be created in "docked" mode. If this is set to false

the window will be a normal window, otherwise it will be created a document docked on creation.
Parameter 'mode': True will allow the window to dock, false will disallow it
returns: Nothing.

saveTableToFile(fileName:String)
Summary: Saves the whole content of the table into a specified file
Parameter 'fileName': The filename to save to
returns:

saveTableToFileWriter(fileWriter:Plugin)
Summary: Saves the whole content of the table into a fileWriter
Parameter 'fileWriter': The filewriter to save into
returns:

saveRowToFileWriter(fileWriter:Plugin)
Summary: Saves the currently active row into the fileWriter
Parameter 'fileWriter': The filewriter to use
returns:

activateAutosave(filename:String interval:Integer)
Summary: Activates the autosave funktion of the table, will save periodically into the specified file
Parameter 'filename': The filename to save to
Parameter 'interval': The interval between saves in miliseconds
returns: Does not return anything

deactivateAutosave()
Summary: Deactivates the autosaving feature in case its no longe wanted. If the autosave was active this will also make the feature save one final dump of the table contents into the specified filename. This is also automatically called whenever the program stops execution
returns:

Events

windowClosed
*description:*Fires when the window is closed
*parameters:*unkown parameter types

Plugin PlgSimpleSocketServer Version: (3.2 build 3578)

This plugin provides a simple socket server. In comparison to a "real" socket server, only one client connection is allowed at any time. This greatly simplifies the communication between client and server and should be sufficient for most scenarios.

Methods

open(port:Integer)
Summary: Opens a socket connection to the specified port and tcp / ip adress Will not return to execution until a connection is established
Parameter 'port': The port the server is to listen to

returns:

close()
Summary: Closes the socket connection
returns:

open()
Summary: Opens the socket with previously supplied address and port
returns:

setAddress(address:String)
Summary: Sets the IPAdress or hostname to be used by the common socket
Parameter 'address': The adress to be used
returns:

setPort(port:Integer)
Summary: Sets the port to be used by a later open command
Parameter 'port':
returns:

open(addr:String port:Integer)
Summary: Opens the socket with supplied adress and port
Parameter 'addr':
Parameter 'port':
returns:

setNewLine(newLine:String)
Summary: Sets the markers for a new line. Usually this will be \r for unix like clients, and \r\n for windows clients. Note that setNewLineUnix and setNewLineWindows can be used.
Parameter 'newLine': Any string that marks the beginning of a new line
returns:

setNewLineUnix()
Summary: Sets the marker for a new line to unix-friendly \r only.
returns:

setNewLineWindows()
Summary: Sets the marker for a new line to Windows(tm) friendly \r\n.
returns:

setReadTimeOut(timeout:Integer)
Summary: Sets the Read Timeout for Serial operations
Parameter 'timeout': The timeout for the reading operation in seconds

writeLine(str:String)
Summary: Sends a string to the Com Component, followed by carriage return linefeed
Parameter 'str': The String to be send
returns:

write(str:String)
Summary: Sends a string to the Com Component, no carriage return linefeed
Parameter 'str': The String to be send
returns:

writeByte(by:Integer)
Summary: Writes a single byte into the comport
Parameter 'by': The byte to be written
returns:

writeWord(word:Integer)
Summary: Writes a single word into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The word to be written
returns:

writeLong(by:Integer)
Summary: Writes a single longword into the comport, olcus will automatically convert the value if it is in range
Parameter 'by': The long to be written
returns:

readLineAsync()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readline()
Summary: Returns null if there was no line of text available, or a line of text if one is there
returns: null or a line of text, encoded using the local encoding scheme

readByteAsync()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readByte()
Summary: Reads a single byte from the serial interface, async (does not wait for the byte) Will return -1 if no byte is available
returns:

readWordAsync()
Summary: Reads a single word from the serial interface
returns:

readWord()
Summary: Reads a single word from the serial interface
returns:

readLongAsync()
Summary: Reads a single longword from the serial interface
returns:

readLong()
Summary: Reads a single longword from the serial interface
returns:

flush()
Summary: Makes the serial device flush the buffer, use only when autoflush is disabled.
returns:

setAutoFlush(auto:Boolean)
Summary: Enables or disables the autoflush feature. True will make the serial connection automatically flush all written Data, False will require the manual flushing via the .flush() method.
Parameter 'auto': True or False
returns:

Events

getdata
*description:*fires when new Data is available, provides data as a String.
*parameters:*OString
getRaw
*description:*fires when new Data is available, provides data as a Byte array.
*parameters:*OArray
getLine
*description:*fires when a line of text is available, the line end marker is determined by .setNewLine*. Line is Provided as String.
*parameters:*OString

Plugin PlgFile Version: (3.2 build 3578)

The FileWriter plugins allows to write data into a textfile. One Plugin needs to be instanced for every textfile to write data into.

Methods

exists(path:String)
Summary: Returns true if a file exists
Parameter 'path': The path to the file to be checked
returns:

createSequencedFileName(path:String extention:String)
Summary: Creates a sequenced filename. If you provide a path like "c:\temp\data" and extension "txt" It will first create c:\temp\data01.txt and on the second run c:\temp\data02.txt and so forth.
Parameter 'path': The path element of the filename
Parameter 'extention': The desired filename extension

returns:

createSequencedFileName(path:String)
Summary: Creates a sequenced filename. If you provide a path like "c:\temp\data" and extension "txt" It will first create c:\temp\data01.txt and on the second run c:\temp\data02.txt and so forth.
Parameter 'path': The path element of the filename
Parameter 'extention': The desired filename extension
returns:

deleteFile(path:String)
Summary: Deletes an existing file
Parameter 'path': The path to the file
returns:

fileOpenDialog(title:String)
Summary: Displays a file open dialog, allowing the user to select a file for opening. If the user selects cancel or otherwise closes the dialog, an empty string "" will be returned, otherweise the full filepath is returned
Parameter 'title': The title to be displayed in the dialog
returns:

fileSaveDialog(title:String)
Summary: Displays a file open dialog, allowing the user to select a file for opening. If the user selects cancel or otherwise closes the dialog, an empty string "" will be returned, otherweise the full filepath is returned
Parameter 'title': The title to be displayed in the dialog
returns:

Plugin PlgADO Version: (0.5)

The ADO Plugin allows access to Various Databases via the Microsoft ADO.Net framework While only part of ADO's funktionality is exposed to the script, all essential database operations should be possible using this plugin. Contact Adminstration if you require direct MySQL or Oracle access. (ODBC will work fine)

Methods

openMSSql(connectionString:String)
Summary: Opens a connection to Microsoft SQL Server
Parameter 'connectionString': The connectionsstring that provides details for connecting to the target database
returns:

openXLS(filename:String)
Summary: Opens an ODBC connection to an Excel table, convinience method, can be done via open ODBC, too.
Parameter 'filename': Filename of the excel table
returns:

openODBC(connectionString:String)
Summary: Opens an ODBC Connection using the specified Connection String
Parameter 'connectionString':
returns:

select(selectStatement:String)
Summary: Executes a select statement from a database
Parameter 'selectStatement': The select statement
returns: Returns the number of fetched rows

execute(sqlStatement:String)
Summary: Executes an SQL Statement on the open connection (INSERT, SET)
Parameter 'sqlStatement': The SQL Statement
returns: Returns the number of fetched rows

selectAllFromTable(table:String)
Summary: Executes a select statement from a database
Parameter 'selectStatement': The select statement
returns: Returns the number of fetched rows

userSelectRow()
Summary: Allows the user to select a row from the currently active database table
returns:

setCurrentRow(idx:Integer)
Summary: Sets the currently active row by an index
Parameter 'idx': The index to the row, starting with 0
returns:

getCurrentRowIndex()
Summary: Returns the index to the current row
returns:

hasNext()
Summary: Returns true, if there is another row to navigate to with the nextRow command.
returns:

hasPrevious()
Summary: Returns true, if there is another row to navigate to with the nextRow command.
returns:

nextRow()
Summary: Sets the row pointer to the next available row
returns:

lastRow()
Summary: Returns the pointer to the previous row

returns:

firstRow()
Summary: Sets the row cursor to the very first row
returns:

getRowColumn(idx:Integer)
Summary: Returns the contence of a column
Parameter 'idx': Index to the colmn starting with 0 for the first
returns:

getRowColumn(name:String)
Summary: Returns the contence of a column, indexed by the name of the colmn
Parameter 'name':
returns:

setRowColumn(name:String value:String)
Summary: Sets data in the active row
Parameter 'name': Name of the column
Parameter 'value': The value to be set
returns:

setRowColumn(index:Integer value:String)
Summary: Sets data in the actuve row
Parameter 'index': Index to the column, starting with index 0
Parameter 'value': The value to be set
returns:

insertRow()
Summary: Creates a new row into the dataset, to be written into
returns:

commit()
Summary: Commits any changes to the database
returns:

close()
Summary: Closes the database connection
returns:

Plugin PlgBioOpViewer Version: (1.0)

Experimental Plugin for communication with the BioObserve Viewer Application The plugin needs an Identifier to communicate with incoming plugin connections

Methods

waitForConnection()

Summary: This method will wait until a connection is made by the corresponding client
returns: Returns nothing, but will continue execution once the connection is made

setIdentifier(id:String)
Summary: Sets the identifier for the Plugin (the ID that the client needs to provide to attach to this plugin)
Parameter 'id': The id to set, any String
returns: nothing

Events

receiveLine
*description:*fires when a Line is received
*parameters:*OString

Plugin PlgAnimalIdentificator Version: (1.0)

This plugin provides methods to utilize an AnimalIdentificator hardware device.

Methods

setRFIDToDatamars()
Summary: Sets the RFID detector in datamars mode
returns:

setRFIDToDatamars(id:Integer)
Summary: Sets the RFID detector in datamars mode
returns:

setRFIDToTrovan()
Summary: Sets the RFID detector in trovan mode
returns:

setRFIDToTrovan(id:Integer)
Summary: Sets the RFID detector in trovan mode
returns:

getVersion(id:Integer)
Summary: Returns the version of the AnimalIdentificator (not yet ;-) just asks for it
Parameter 'id': hardware CANid
returns:

setCanAddress(addr:Integer)
Summary: Sets the Adress of the Can-Device to be used, by default, all incoming traffic will be processed but once an id is set here, only the traffic of the specified device will be processed. If this value is set to -1, all traffic will be processed.
Parameter 'addr': The address to set (0-7)
returns:

setCanDriver(canDriver:Plugin)
Summary: Connect this plugin with its communication driver for door operation and movement detection, For example a comport plugin.
Parameter 'comPlg': The comport plugin
returns:

Events

dataReceived
*description:*fired when data is received. Parameters are: timestamp, device id, antenna id, data as string.
*parameters:*OInteger OInteger OInteger OString

Plugin FBIRfidReader Version: (1.0)

A plugin to use the FBI-Rfid-Reader device, that connects via Can-Bus. For this plugin the setAddr parameter is mandatory

Methods

setCanDriver(canDriver:Plugin)
Documentation unavailable
setCanAddress(addr:Integer)
Summary: This sets the address for the can device. Also it will trigger initialization
Parameter 'addr':
returns:

getHardwareVersion()
Summary: Will request the version information from the device.
returns:

getLastTag()
Summary: Asks the device to report the last scanned tag number
returns:

setFilters(device:Integer lo:Boolean hi:Boolean)
Summary: Sets the main low-pass and high-pass cut-off frequency.
Parameter 'lo': low-pass True : 6 kHz False: 3 kHz
Parameter 'hi': True : 160 kHz False: 40 kHz
Parameter 'device': The SubDevice number to set this parameter for
returns:

getFilterLow(device:Integer)
Summary: Reports the main low-pass cut-off frequency.
Parameter 'device': The SubDevice number to request his parameter from
returns:

getFilterHigh(device:Integer)
Summary: Reports the main high-pass cut-off frequency.

Parameter 'device': The SubDevice number to request his parameter from
returns:

setAmplifierGain(device:Integer zero:Boolean one:Boolean)
Summary: Sets amplifier gain factors
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'zero': Gain 0 True : gain 32 False: gain 16
Parameter 'one': Gain 1 True : gain 31.5 False: gain 6.22
returns:

getAmplifierGain0(device:Integer)
Summary: Reports Sets amplifier gain0 factor
Parameter 'device': The SubDevice number to request his parameter from
returns:

getAmplifierGain1(device:Integer)
Summary: Reports Sets amplifier gain0 factor
Parameter 'device': The SubDevice number to request his parameter from
returns:

setDisableCoilDriver(device:Integer bo:Boolean)
Summary: disables the coil driver
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'bo': True : coil driver inactive False: coil driver active
returns:

getCoilDriverDisabled(device:Integer)
Summary: Reports coil driver status
Parameter 'device': The SubDevice number to request his parameter from
returns:

setComperatorHysteresis(device:Integer bo:Boolean)
Summary: sets the data comparator hysteresis
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'bo': True : hysteresis ON False: hysteresis OFF
returns:

getHysteresisStatus(device:Integer)
Summary: Reports is comparator Hysteresis is enabled
Parameter 'device': The SubDevice number to request his parameter from
returns:

setPowerDown(device:Integer bo:Boolean mode:Boolean)
Summary: Enabled Power-down mode
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'bo': True : device power-down False: device active
Parameter 'mode': True : Power-down mode False: Idle mode
returns:

getPowerDownEnabled(device:Integer)
Summary: Reports if powerdown is enabled
Parameter 'device': The SubDevice number to request his parameter from
returns:

getPowerDownMode(device:Integer)
Summary: Reports if powerdown mode
Parameter 'device': The SubDevice number to request his parameter from
returns:

setFreeze(device:Integer frz:Integer)
Summary: facility to achieve fast settling times (MSB and LSB)
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'frz': 0 : normal operation 1 : main low-pass is frozen; main high-pass is pre-charged to level on pin QGND 2 : main low-pass is frozen; time constant of main high-pass is reduced by a factor of 16 for bit FILTERH = 0 and by a factor of 8 for bit FILTERH = 1 3 : main high-pass time constant is reduced by a factor of 16 for bit FILTERH = 0 and by a factor of 8 for bit FILTERH = 1; second high-pass is pre-charged
returns:

getFreeze(device:Integer)
Summary: Reports freeze mode
Parameter 'device': The SubDevice number to request his parameter from
returns:

setAcqAmp(device:Integer bo:Boolean)
Summary: store signal amplitude
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'bo': True : store actual amplitude of the data signal as reference for later amplitude comparison False: set status bit AMPCOMP when the actual data signal amplitude is higher than the stored reference
returns:

getAcqAmp(device:Integer)
Summary: Reports store signal amplitude feature is enabled
Parameter 'device': The SubDevice number to request his parameter from
returns:

setThReset(device:Integer bo:Boolean)
Summary: reset threshold generation of digitizer
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'bo': True : reset False: no reset
returns:

getThReset(device:Integer)
Summary: Reports if reset threshold generation of digitizer is enabled
Parameter 'device': The SubDevice number to request his parameter from

returns:

setClockFreq(device:Integer fsel:Integer)
Summary: clock frequency selection (MSB and LSB)
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'fsel': 0 : 4 MHz 1 : 8 MHz 2 : 12 MHz 3 : 16 MHz
returns:

getClockFreq(device:Integer)
Summary: Reports the clock frequency
Parameter 'device': The SubDevice number to request his parameter from
returns:

disableSmartComperator(device:Integer bo:Boolean)
Summary: disable smart comparator
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'bo': True : smart comparator: off False: smart comparator: on
returns:

getSmartComperatorDisabled(device:Integer)
Summary: Reports if rdisable smart comparator is disabled
Parameter 'device': The SubDevice number to request his parameter from
returns:

disableLowPass1(device:Integer bo:Boolean)
Summary: disable Low pass 1
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'bo': True : low-pass: on False: low-pass: off
returns:

getLowPassDisabled(device:Integer)
Summary: Reports if disable Low pass 1 is disabled
Parameter 'device': The SubDevice number to request his parameter from
returns:

setSamplingTime(device:Integer samplingTime:Integer)
Summary: Sets the Sampling Time
Parameter 'device': The SubDevice number to set this parameter for
Parameter 'samplingTime': Sampling Time
returns:

getSamplingTime(device:Integer)
Summary: Returns the sampling time set for the specified device
Parameter 'device': The SubDevice number to request his parameter from
returns:

getReadingDelay(devid:Integer)
Summary: Gets reading delay (the minimum delay between two readings of RFIDs)

Parameter 'devid': Sub Device ID
returns:

setReadingDelay(devid:Integer delay:Integer)
Summary: Sets the Reading Delay, the minimum delay between two RFID readings
Parameter 'devid': The subdevice ID to set the delay for
Parameter 'delay': The delay to set to in miliseconds (0-25500). Steps by 100.
returns:

getConfigView(devID:Integer)
Summary: Shows configuration overview of selected device
Parameter 'devID': Sub-Device ID
returns:

getConfigViewDebug(devID:Integer)
Summary: Shows configuration overview of selected device, with true / false values for debugging
Parameter 'devID': Sub-Device ID
returns:

setRFIDToDatamars()
Summary: Sets the RFID detector in datamars mode
returns:

setRFIDToDatamars(id:Integer)
Summary: Sets the RFID detector in datamars mode
returns:

setRFIDToTrovan()
Summary: Sets the RFID detector in trovan mode
returns:

setRFIDToTrovan(id:Integer)
Summary: Sets the RFID detector in trovan mode
returns:

getVersion(id:Integer)
Summary: Returns the version of the AnimalIdentificator (not yet ;-) just asks for it
Parameter 'id': hardware CANid
returns:

Events

antfail
*description:*is triggered then the Antenna Fail signal is received from one of the reader Antenna. Parameter is: Antenna ID, Status (true = ok, false = failed)
*parameters:*OInteger OBoolean
dataReceived
*description:*fired when data is received. Parameters are: timestamp, device id, antenna id, data as

string.
*parameters:*OInteger OInteger OInteger OString

Plugin PlgMouseWheel Version: (1.0)

The mousewheel plugin provides functionality to use a mousewheel hardware device. The plugin needs an operational CanBus plugin for its operation, that needs to be passed via the method "setCanDriver" The two mousewheels are enumerated by 1 and 2. Meaning that every function that references a mousewheel will use either index 1 or 2. Events will also identify this mousewheel index.

Methods

setParameters(mwIndex:Integer diameter:Integer ticksPerTurn:Integer sendTIME:Integer measureTIME:Integer scanTIME:Integer)
Summary: Sets the Parameters for one of the MouseWheel Devices
Parameter 'mwIndex': The index to the mousewheel device. 1-2 (can id 4, can id 5)
Parameter 'diameter': The wheel Diameter
Parameter 'ticksPerTurn': The ticks per turn
Parameter 'sendTIME': Send Time
Parameter 'measureTIME': Measure Time
Parameter 'scanTIME': Scan Time
returns:

resetSpeedCalc(mwIndex:Integer)
Summary: Resets the Calculation of the Average and the Maximum Speed Should be called prior to any measurement
Parameter 'mwIndex': The index to the mousewheel device. 1-2 (can id 4, can id 5)
returns: Does not return anything

getCurrentAverageSpeed(mwIndex:Integer)
Summary: Returns the average speed for this wheel, passed through as a double. resetSpeedCalc should be called before starting to take measurements
Parameter 'mwIndex': The index to the mousewheel device. 1-2 (can id 4, can id 5)
returns:

getCurrentMaximumSpeed(mwIndex:Integer)
Summary: Returns the maximum speed for this wheel resetSpeedCalc should be called before starting to take measurements
Parameter 'mwIndex': The index to the mousewheel device. 1-2 (can id 4, can id 5)
returns:

getCurrentMinimumSpeed(mwIndex:Integer)
Summary: Returns the minimum speed for this wheel resetSpeedCalc should be called before starting to take measurements
Parameter 'mwIndex': The index to the mousewheel device. 1-2 (can id 4, can id 5)
returns:

getCurrentCount(mwIndex:Integer)

Summary: Returns the number of clicks since resetSpeedCalc was last called.
Parameter 'mwIndex': The index to the mousewheel device. 1-2 (can id 4, can id 5)
returns:

getTotalCounts(mwIndex:Integer)
Summary: Returns the number of clicks since the Device was initialized
Parameter 'mwIndex': The index to the mousewheel device. 1-2 (can id 4, can id 5)
returns:

getCurrentSpeed(mwIndex:Integer)
Summary: Returns the current speed as it was last reported by the device
Parameter 'mwIndex': The index to the mousewheel device. 1-2 (can id 4, can id 5)
returns:

setCanAddress(addr:Integer)
Summary: Sets the Adress of the Can-Device to be used, by default, all incoming traffic will be processed but once an id is set here, only the traffic of the specified device will be processed. If this value is set to -1, all traffic will be processed.
Parameter 'addr': The address to set (0-7)
returns:

setCanDriver(canDriver:Plugin)
Summary: Connect this plugin with its communication driver for door operation and movement detection, For example a comport plugin.
Parameter 'comPlg': The comport plugin
returns:

Events

wheelData
*description:*fired when data from the Wheel is received. id, ms_wcnt, ms_speed, ticsturn, diameter, direction.
*parameters:*OInteger OInteger OInteger OInteger OInteger OInteger

Plugin PlgCanBusAnimalIdentificatorSim Version: (1.1)

Simulates a can bus, for debugging purposes. A window is shown that shows all data send to the canbus. Responses can be simulated by the user.

Methods

sendData(id:Integer data1:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
returns:

sendData(id:Integer data1:Integer data2:Integer)
Summary: Sends data to the Canbus.

Parameter 'id': The identifier
Parameter 'data1': The data byte 1
Parameter 'data2': The data byte 2
returns:

sendData(id:Integer data1:Integer data2:Integer data3:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
Parameter 'data2': The data byte 2
Parameter 'data3': The data byte 3
returns:

sendData(id:Integer arr:Array)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'arr': An array containing the bytes to send cleanly indexed by 0,1,2,...
returns:

sendDataHex(id:Integer data:String)
Summary: Sends data to the CanBus, encodet in hexadecimal like "CAFFEE"
Parameter 'id': The identifier
Parameter 'data':
returns:

sendDataStr(id:Integer sData:String)
Summary: Sends a string to the canbus interface
Parameter 'id': The id to send by
Parameter 'sData': The data to send
returns:

setDevId(id:Integer)
Summary: Sets the device ID of this canbus interface
Parameter 'id': The ID of the canbus device to be used
returns:

setDebug(debug:Boolean)
Summary: Sets this can-Bus adapter to DebugMode. In debug mode all received and send can messages will be reported to the console as a warning. Useful for debugging purposes.
Parameter 'debug': True means all can messages are reported, False means they are not.
returns:

Events

canMessage
*description:*fired when a canbus message is received. The message is returned as string.
*parameters:*OString
canError

*description:*fired when an error state is received. Error is returned as string.
*parameters:*OString
dataReceived
*description:*is fired when new data is received. Contains timestamp, id, datalen, data as byte array (zero based).
*parameters:*OInteger OInteger OInteger OArray

Plugin PlgCanBusIXXAT Version: (1.1)

Can class plugin, allows communication with an ixxat canbus device

Methods

sendData(id:Integer data1:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
returns:

sendData(id:Integer data1:Integer data2:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
Parameter 'data2': The data byte 2
returns:

sendData(id:Integer data1:Integer data2:Integer data3:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
Parameter 'data2': The data byte 2
Parameter 'data3': The data byte 3
returns:

sendData(id:Integer arr:Array)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'arr': An array containing the bytes to send cleanly indexed by 0,1,2,...
returns:

sendDataHex(id:Integer data:String)
Summary: Sends data to the CanBus, encodet in hexadecimal like "CAFFEE"
Parameter 'id': The identifier
Parameter 'data':
returns:

sendDataStr(id:Integer sData:String)
Summary: Sends a string to the canbus interface
Parameter 'id': The id to send by

Parameter 'sData': The data to send
returns:

setDevId(id:Integer)
Summary: Sets the device ID of this canbus interface
Parameter 'id': The ID of the canbus device to be used
returns:

setDebug(debug:Boolean)
Summary: Sets this can-Bus adapter to DebugMode. In debug mode all received and send can messages will be reported to the console as a warning. Useful for debugging purposes.
Parameter 'debug': True means all can messages are reported, False means they are not.
returns:

Events

canMessage
*description:*fired when a canbus message is received. The message is returned as string.
*parameters:*OString
canError
*description:*fired when an error state is received. Error is returned as string.
*parameters:*OString
dataReceived
*description:*is fired when new data is received. Contains timestamp, id, datalen, data as byte array (zero based).
*parameters:*OInteger OInteger OInteger OArray

Plugin PlgCanBusFBI Version: (1.1)

Canbus plugin that allows communication via an FBI Canbus device.

Methods

open(rsPort:String)
Summary: Defines the virtual RS232 port to be used to connect with the FBI Canbus interface
Parameter 'rsPort': The port to be used
returns:

sendData(id:Integer data1:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
returns:

sendData(id:Integer data1:Integer data2:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
Parameter 'data2': The data byte 2

returns:

sendData(id:Integer data1:Integer data2:Integer data3:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
Parameter 'data2': The data byte 2
Parameter 'data3': The data byte 3
returns:

sendData(id:Integer arr:Array)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'arr': An array containing the bytes to send cleanly indexed by 0,1,2,...
returns:

sendDataHex(id:Integer data:String)
Summary: Sends data to the CanBus, encodet in hexadecimal like "CAFFEE"
Parameter 'id': The identifier
Parameter 'data':
returns:

sendDataStr(id:Integer sData:String)
Summary: Sends a string to the canbus interface
Parameter 'id': The id to send by
Parameter 'sData': The data to send
returns:

setDevId(id:Integer)
Summary: Sets the device ID of this canbus interface
Parameter 'id': The ID of the canbus device to be used
returns:

setDebug(debug:Boolean)
Summary: Sets this can-Bus adapter to DebugMode. In debug mode all received and send can messages will be reported to the console as a warning. Useful for debugging purposes.
Parameter 'debug': True means all can messages are reported, False means they are not.
returns:

Events

canMessage
*description:*fired when a canbus message is received. The message is returned as string.
*parameters:*OString
canError
*description:*fired when an error state is received. Error is returned as string.
*parameters:*OString
dataReceived

*description:*is fired when new data is received. Contains timestamp, id, datalen, data as byte array (zero based).
*parameters:*OInteger OInteger OInteger OArray

Plugin PlgCanBusSim Version: (1.1)

Simulates a can bus, for debugging purposes. A window is shown that shows all data send to the canbus. Responses can be simulated by the user.

Methods

sendData(id:Integer data1:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
returns:

sendData(id:Integer data1:Integer data2:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
Parameter 'data2': The data byte 2
returns:

sendData(id:Integer data1:Integer data2:Integer data3:Integer)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'data1': The data byte 1
Parameter 'data2': The data byte 2
Parameter 'data3': The data byte 3
returns:

sendData(id:Integer arr:Array)
Summary: Sends data to the Canbus.
Parameter 'id': The identifier
Parameter 'arr': An array containing the bytes to send cleanly indexed by 0,1,2,...
returns:

sendDataHex(id:Integer data:String)
Summary: Sends data to the CanBus, encodet in hexadecimal like "CAFFEE"
Parameter 'id': The identifier
Parameter 'data':
returns:

sendDataStr(id:Integer sData:String)
Summary: Sends a string to the canbus interface
Parameter 'id': The id to send by
Parameter 'sData': The data to send
returns:

setDevId(id:Integer)
Summary: Sets the device ID of this canbus interface
Parameter 'id': The ID of the canbus device to be used
returns:

setDebug(debug:Boolean)
Summary: Sets this can-Bus adapter to DebugMode. In debug mode all received and send can messages will be reported to the console as a warning. Useful for debugging purposes.
Parameter 'debug': True means all can messages are reported, False means they are not.
returns:

Events

canMessage
*description:*fired when a canbus message is received. The message is returned as string.
*parameters:*OString
canError
*description:*fired when an error state is received. Error is returned as string.
*parameters:*OString
dataReceived
*description:*is fired when new data is received. Contains timestamp, id, datalen, data as byte array (zero based).
*parameters:*OInteger OInteger OInteger OArray

Plugin FBIInterface Version: (1.0)

Implements serial communication with the fbi serial interface

Methods

setComPort(port:String)
Documentation unavailable
setCanBus(can:Plugin)
Documentation unavailable
ping()
Summary: Sends a ping to the device to check if the basic funktionality is in place
returns:

Plugin PlgWindowDisplay Version: (1.0)

The window Display plugin allows to display the Output of a GFX Plugin locally in a window. Usually for reference, this plugin supports all features of a display plugin.

Methods

setDeviceID(id:String)
Summary: Sets the DeviceID to be used when events are signaled from this display
Parameter 'id': The id to be used by this device

clearFilter(typ:String)
Summary: Clears the specified filter, and resets the mode to default. (blacklist)
Parameter 'type': The filter type to control, for example, 'device', 'area' or 'type'
returns:

addFilterItem(typ:String item:String)
Summary: Adds a filter item to this displays filter. Filters control, what kind of pickevents are reported when they occur. Different filter types control different aspects of pick events. For example: 'device' : Controls which devices are allowed to send pick events. Usually it makes most sence to control these via the GFX plugin. 'area' : Controls which areas are allowed to generate picks. Using the filter in whitelist mode and passing "id1" as an item, for example will block all picks that are not generated within the areas which have the ide "id1" 'type' : Controls the type of picks that are to be reported. Using the filter in blacklist mode (default) and passing the item "1" for example, will block all "move" events, which is often required.
Parameter 'type': The filter type to control, for example, 'device', 'area' or 'type'
Parameter 'item': The item to be added to the filtes list
returns:

setFilterMode(typ:String mode:String)
Summary: Sets the mode the filter operates in. Two modes are available: 'blacklist' : When the Filter operates in blacklist mode, all items that are contained in the filter will be ignored. 'whitelist' : When the filter operates in whitelist mode, all items not contained in the filter will be ignored.
Parameter 'typ': The filter type to control, for example, 'device', 'area' or 'type'
Parameter 'mode': The mode to set the filter to, either 'blacklist' or 'whitelist'
returns:

Events

pick
*description:*fired when a pick occurs. Parameters are deviceID, areaID, timestamp, type (0 = mouseUp, 1 = mouseDown, 2 = mouseMove), x, y.
*parameters:*OString OString OInteger OInteger OInteger OInteger

Plugin PlgGfx Version: (1.0)

This plugin provides functions for creating graphical stimuli. Several display Plugins can be "Added" to this plugin, and all graphical commands send to this plugin will be shown on all connected displays. Displays do not neccessarily have to reside on the Host machine, they might be running on a different computer and communicate with the main class via socket connection, or RS232.

Methods

setDisplay(display:Plugin)
Summary: Assigns a Display to this graphics class
Parameter 'display':
returns:

show()
Summary: Initializes all Displays and makes them visible
returns: Nothing

hide()
Summary: Hides all displays, making them invisible
returns: Nothing

setPosition(x:Integer y:Integer)
Summary: Sets the display position to the desired coordinates.
Parameter 'x': The x-coordinate for the display
Parameter 'y': The y-coordinate for the display
returns:

setSize(w:Integer h:Integer)
Summary: Sets the displays height
Parameter 'w': Width of the display
Parameter 'h': Height of the display
returns:

drawLine(x1:Integer y1:Integer x2:Integer y2:Integer)
Summary: Draws a line from the starting point to the endpoint
Parameter 'x1': X-Coordinate of the starting point
Parameter 'y1': Y-Coordinate of the starting point
Parameter 'x2': X-Coordinate of the end point
Parameter 'y2': Y-Coordinate of the end point
returns:

drawRect(x1:Integer y1:Integer w:Integer h:Integer)
Summary: Draws a rectangle from the starting point with the specified width and height
Parameter 'x1': X-Coordinate of the starting point
Parameter 'y1': Y-Coordinate of the starting point
Parameter 'w': Width of the Rectangle
Parameter 'h': Height of the Rectangle
returns:

drawEllipse(x1:Integer y1:Integer w:Integer h:Integer)
Summary: Draws an ellipse from the starting point with the specified width and height
Parameter 'x1': X-Coordinate of the starting point
Parameter 'y1': Y-Coordinate of the starting point
Parameter 'w': Width of the Rectangle
Parameter 'h': Height of the Rectangle
returns:

drawCircle(x1:Integer y1:Integer r:Integer)
Summary: Draws a circle, with the specified radius
Parameter 'x1': Upper left corner of a rectangle around the circle
Parameter 'y1': Upper left corner of a rectangle around the circle

Parameter 'r': The circle's radius
returns:

bufferImage(alias:String path:String)
Summary: Buffers an image in memory. The image will be stored under an alias, making it easy to reference
Parameter 'alias': The alias to store the image under
Parameter 'path': The physical path to the image
returns:

drawImage(alias:String x:Integer y:Integer w:Integer h:Integer)
Summary: Draws an image to the canvas
Parameter 'alias': The alias or the physical path to the image
Parameter 'x': The x coordinate of the upper left corner of the image
Parameter 'y': The y coordinate of the upper left corner of the image
Parameter 'w': The width of the image to be drawn
Parameter 'h': The height of the image to be drawn
returns:

clearClickAreas()
Summary: Clears all previously defined clickareas
returns:

addClickArea(id:String x:Integer y:Integer width:Integer height:Integer)
Summary: Defines a click area. When a click occures inside of the defined area the corresponding id will be returned along with the click event. Clicks outside of any clickarea are reported with the default ID "Screen". Multiple click areas may share the same id.
Parameter 'id': The ID to be reported for clicks inside of the defined area
Parameter 'x': The x coordinate of the area's upper left corner
Parameter 'y': The y coordinate of the area's upper left corner
Parameter 'width': The width of the area
Parameter 'height': The height of the area
returns:

drawClickAreas()
Summary: This command is for debugging purposes only. It draws all currently existing pickareas using the currently selected pen
returns:

clearFilter(typ:String)
Summary: Clears the specified filter for all connected displays, and resets the mode to default. (blacklist)
Parameter 'type': The filter type to control, for example, 'device', 'area' or 'type'
returns:

addFilterItem(typ:String item:String)
Summary: Adds a filter item to all displays filters. Filters control what kind of pickevents are reported when they occur. Different filter types control different aspects of pick events. For

example: 'device' : Controls which devices are allowed to send pick events. Usually it makes most sence to control these via the GFX plugin. 'area' : Controls which areas are allowed to generate picks. Using the filter in whitelist mode and passing "id1" as an item, for example will block all picks that are not generated within the areas which have the ide "id1" 'type' : Controls the type of picks that are to be reported. Using the filter in blacklist mode (default) and passing the item "1" for example, will block all "move" events, which is often required.
Parameter 'type': The filter type to control, for example, 'device', 'area' or 'type'
Parameter 'item': The item to be added to the filtes list
returns:

setFilterMode(typ:String mode:String)
Summary: Sets the mode the filter operates in. Two modes are available: 'blacklist' : When the Filter operates in blacklist mode, all items that are contained in the filter will be ignored. 'whitelist' : When the filter operates in whitelist mode, all items not contained in the filter will be ignored.
Parameter 'typ': The filter type to control, for example, 'device', 'area' or 'type'
Parameter 'mode': The mode to set the filter to, either 'blacklist' or 'whitelist'
returns:

Events

pick
*description:*fired when a pick occurs. Parameters are deviceID, areaID, timestamp, type (0 = mouseUp, 1 = mouseDown, 2 = mouseMove), x, y.
*parameters:*OString OString OInteger OInteger OInteger OInteger

Plugin PlgRemoteDisplay Version: (1.0)

This Plugin is used to allow an application to communicated with a display that is not running on the same machine as the olcus application. The remote display might be an application running on a PDA device, or just a computer in a different room, connected via ethernet. A communications port needs to be assigend using the method setCommDriver. The most common communications port is likely going to be the SocketClient class.

Methods

setCommDriver(drvr:Plugin)
Summary: Determines which communication port is to be used for communicating with the remote display
Parameter 'drvr': The driver to be used
returns:

setDeviceID(id:String)
Summary: Sets the device ID to be used when events are reported from this display
Parameter 'id': The id to be used by reported events
returns:

clearFilter(typ:String)
Summary: Clears the specified filter, and resets the mode to default. (blacklist)
Parameter 'type': The filter type to control, for example, 'device', 'area' or 'type'

returns:

addFilterItem(typ:String item:String)
Summary: Adds a filter item to this displays filter. Filters control, what kind of pickevents are reported when they occur. Different filter types control different aspects of pick events. For example: 'device' : Controls which devices are allowed to send pick events. Usually it makes most sence to control these via the GFX plugin. 'area' : Controls which areas are allowed to generate picks. Using the filter in whitelist mode and passing "id1" as an item, for example will block all picks that are not generated within the areas which have the ide "id1" 'type' : Controls the type of picks that are to be reported. Using the filter in blacklist mode (default) and passing the item "1" for example, will block all "move" events, which is often required.
Parameter 'type': The filter type to control, for example, 'device', 'area' or 'type'
Parameter 'item': The item to be added to the filtes list
returns:

setFilterMode(typ:String mode:String)
Summary: Sets the mode the filter operates in. Two modes are available: 'blacklist' : When the Filter operates in blacklist mode, all items that are contained in the filter will be ignored. 'whitelist' : When the filter operates in whitelist mode, all items not contained in the filter will be ignored.
Parameter 'typ': The filter type to control, for example, 'device', 'area' or 'type'
Parameter 'mode': The mode to set the filter to, either 'blacklist' or 'whitelist'
returns:

Events

pick
*description:*fired when a pick occurs. Parameters are deviceID, areaID, timestamp, type (0 = mouseUp, 1 = mouseDown, 2 = mouseMove), x, y.
*parameters:*OString OString OInteger OInteger OInteger OInteger

Plugin PlgGUI Version: (3.1)

The gui Plugin is used to create a user interface that an olcus.net user can use to review and modify configuration data. The gui is seperated in "tabs" but each tab can be dragged from the tab container, and used as a separate window. The layout of the tabs can be stored and retrieved, so the layout can be restored after olcus has been restarted.

Methods

addTab(name:String id:String)
Summary: Creates a tab on the Gui Page
Parameter 'name': Name of the tab as displayed to the user
Parameter 'id': The ID of the tab, as used by the Script
returns:

addTab(name:String)
Summary: Creates a tab on the Gui Page
Parameter 'name': Name of the tab as displayed to the user
returns:

setTab(id:String)
Summary: Selects a specific tab from the tabpage. The tab is referenced either by its id or its name
Parameter 'id': Id or name of the tab. Id should be the prefered referencing method
returns:

setTab(id:Integer)
Summary: Selects a specific tab from the tabpage. The tab is referenced by an index starting with 0
Parameter 'idx': The tabs index
returns:

show()
Summary: Displays the window to the screen Will only work if a seperate window is used. (forceWindowCreation) Will not affect the application main Form.
returns:

hide()
Summary: Hides the Window from displaying. Will only work, if a seperate window was created. (forceWindowCreation) Will not affect the application main form.
returns:

setWindowSize(w:Integer h:Integer)
Summary: Sets the size of the main window
Parameter 'w': Sets the size of the current gui window
Parameter 'h': Sets the height of the current gui window
returns:

setWindowPos(x:Integer y:Integer)
Summary: Sets the position of the gui window, relative to the top left corner of the screen
Parameter 'x': X Position of the window
Parameter 'y': Y Position of said window
returns:

getValue(id:String)
Summary: Returns the value of a user control
Parameter 'id': The control's id
returns:

setValue(id:String val:String)
Summary: Sets the value for a control
Parameter 'id': The control, referenced by its id
Parameter 'val': The value to be set. Please keep in mind that certain controls may only accept certain keywords as value. I.E. checkbox will only accept true or false, while a button will accept nothing.
returns:

addTextBox(text:String id:String value:String)
Summary: Adds a textbox to the currently selected tab page

Parameter 'text': The text, displayed to the user
Parameter 'id': The id
Parameter 'value':
returns:

addCheckBox(text:String id:String value:String)
Summary: Adds a checkbox to the currently selected tab page
Parameter 'text': The text of the checkbox
Parameter 'id': The checkbox' id
Parameter 'value': true, or false, to mark selected, or unselected
returns:

addCheckBox(text:String id:String value:Boolean)
Summary: Adds a checkbox to the currently selected tab page
Parameter 'text': The text of the checkbox
Parameter 'id': The checkbox' id
Parameter 'value': true, or false, to mark selected, or unselected
returns:

addButton(text:String id:String)
Summary: Adds a button to the currently selected tab
Parameter 'text': The buttons's text, displayed to the user
Parameter 'id': The id, to reference the button by, hidden
returns: Nothing

addSeparator(text:String)
Summary: Adds a labeled seperator to the currently selected tab
Parameter 'text': The displayed text and id of the button.
returns:

addSeparator()
Summary: Adds a labeled seperator to the currently selected tab
Parameter 'text': The displayed text and id of the button.
returns:

addRadioGroup(id:String)
Summary: Adds a Radio Group to the currently selected Tab This method only makes sense if more than one option is added via The AddRadioGroupOption Method
Parameter 'id': The radio groups ID
returns:

addRadioGroupOption(id:String optionID:String text:String)
Summary: Adds an option to a radio Group
Parameter 'id': The radio group ID of the radiogroup to add to
Parameter 'optionID': The options id that will be available via value.
Parameter 'text': The text of the option displayed to the user
returns:

addRadioGroupOption(id:String text:String)
Summary: Adds an option to a radio Group
Parameter 'id': The radio group ID of the radiogroup to add to
Parameter 'text': The text of the option displayed to the user, and used as id.
returns:

setPosition(id:String x:Integer y:Integer)
Summary: This method changes the position of a component, the component will then automatically detach from the layout, and has to be reattached if neccessary
Parameter 'id': Id the control is referenced by
Parameter 'x': The x parameter
Parameter 'y': The y parameter

setSize(id:String w:Integer h:Integer)
Summary: The method can change the size of a component, the component will then automatically detach from the automatic layout, and has to be rettached as neccessary
Parameter 'id': Id the control is referenced by
Parameter 'w': The controls width
Parameter 'h': The controls height

setHooks(id:String top:Boolean right:Boolean bottom:Boolean left:Boolean)
Summary: This method will set the provided hooks for a component so it resizes automatically along with the rest of the window
Parameter 'id': The controls id
Parameter 'up': the top anchor
Parameter 'right': the right anchor
Parameter 'bottom': the bottom anchor
Parameter 'left': the left anchor

setEnabled(id:String state:Boolean)
Summary: Enables or disables a control, as selected by the user
Parameter 'id':
Parameter 'state':

setTitle(text:String)
Summary: Sets the title of the GUI object. (The text that is shown in the window)
Parameter 'text': The Title text
returns:

waitForButton(id:String)
Summary: Waits until a certain button inside of the application is clicked
Parameter 'id': The buttons id
returns:

enableLayoutSave(en:Boolean)
Summary: Enables automatic saving of the gui layout or disabled it by default its disabled The programmer is adviced to turn this feaure on before using the show command.
Parameter 'en': true to enable, and false to disable saving of the layout

returns:

forceWindowCreation(en:Boolean)
Summary: Determines if the dockpanel to be used for the gui components can be the main window (if olcus was started in gui mode) or not. If this option is set to true, a new dockpanel window instance will be created whether a main form is available for docking or not.
Parameter 'en': True, means that there will be a new window generated, False means, a new window will only be generated if the form window is not available.
returns:

Events

genericButtonPressed
*description:*fires when a button is pressed. The name of the button is passed as parameter.
*parameters:*OString

Plugin PlgIoWarrior Version: (1.0)

The shared portions of the IOWarrior plugin. Not to be used from within the script. Not possible due to the abstract keyword either.

Methods

open(index:Integer)
Summary: Opens the IOWarrior object by its index. There is no real control about the enumeration of IOWarrior usb objects, so the safer method by serial number is strongly suggested, if more than one IOWarrior is to be used.
Parameter 'index': The IOWarriors index in the enumeration starting with 0
returns:

open(serNo:String)
Summary: Opens the IOWarrior object indexed by its serial number, therefore making sure that the right IOWarrior interface is referenced
Parameter 'serNo': The IOWarriors serial number
returns:

getPin(pinidex:Integer)
Summary: Returns a pin's status.
Parameter 'pinidex': The index of the pin to check
returns:

getPin(literal:String)
Summary: Returns a pin's status
Parameter 'literal': The pin's name, or literal
returns:

setPin(inindex:Integer state:Boolean)
Summary: Sets the pin indicated by the index to the desired state, which is either true or false
Parameter 'inindex': The index to the pin

Parameter 'state': The state to set the pin to
returns:

setPin(literal:String state:Boolean)
Summary: Sets the pin indicated by the used literal to the desired state
Parameter 'literal': The literal to be used
Parameter 'state':
returns:

setLiteralTranslation(lit:String pin:Integer)
Summary: Sets up a literal translator for a pin. Often names are easier to handle than Pin numbers. A feeder controled by pin 12 can be easier handled setLiteralTranslation("feeder",12) is used, so you can access pin 12 by setPin("feeder",true) instead of setPin(12,true) Pins can by design only be assigned to a single literal. Assigning a pin a second literal will overwrite the previous assignment. Literals need to be unique. Assigning the same literal to another pin will revoke the previous assignment.
Parameter 'lit': The literal you want the pin to be accessable under
Parameter 'pin': The pin to create a translation for
returns:

reportTranslation()
Summary: Reports the pin translation table, for debugging purposes The table is returned as literal:pin and is comma separated
returns: Reports the list of translated pins

maskPin(pin:Integer state:Boolean)
Summary: Enabled or disables the masking for a pin
Parameter 'pin': The pin to be masked
Parameter 'state': True will make the plugin ignore all state changes for this pin, false will allow state changes to be reported
returns:

maskPin(literal:String state:Boolean)
Summary: Enabled or disables the masking for a pin
Parameter 'literal': The literal of the pin to be masked
Parameter 'state': True will make the plugin ignore all state changes for this pin, false will allow state changes to be reported
returns:

makeOutput(pin:Integer state:Boolean)
Summary: Defines a pin as an output pin, or revokes output status
Parameter 'pin': The pin to be set as an output pin
Parameter 'state': If state is set to true, the pin will be defined an output pin, if its set to false the output status is revoked
returns:

makeOutput(pin:String state:Boolean)
Summary: Defines a pin as an output pin, or revokes output status

Parameter 'pin': The pin to be set as an output pin
Parameter 'state': If state is set to true, the pin will be defined an output pin, if its set to false the output status is revoked
returns:

lowlevel_write(port:Integer data:Array)
Summary: Writes data to the IOWarrior device
Parameter 'port': The port to write to, please refer to IOWarrior documentation
Parameter 'data': The data bytes to send
returns: Returns the number of written bytes

Events

litPinChanged
*description:*fires when a single pin has changed. Parameters: changed pin name, state of the pin.
*parameters:*OString OBoolean
pinsChanged
*description:*fires when pins have changed. All current pin states are reported as bitmap.
*parameters:*OInteger
pinChanged
*description:*fires when a single pin has changed. Parameters: changed pin no., state of the pin.
*parameters:*OInteger OBoolean

Plugin PlgIoWarriorSim Version: (1.0)

This plugin simulates an IOWarrior device, for debugging purposes. All data send to the Virtual IOWarrior is displayed, while the user can simulate returned data.

Methods

open(index:Integer)
Summary: Opens the IOWarrior object by its index. There is no real control about the enumeration of IOWarrior usb objects, so the safer method by serial number is strongly suggested, if more than one IOWarrior is to be used.
Parameter 'index': The IOWarriors index in the enumeration starting with 0
returns:

open(serNo:String)
Summary: Opens the IOWarrior object indexed by its serial number, therefore making sure that the right IOWarrior interface is referenced
Parameter 'serNo': The IOWarriors serial number
returns:

getPin(pinidex:Integer)
Summary: Returns a pin's status.
Parameter 'pinidex': The index of the pin to check
returns:

getPin(literal:String)

Summary: Returns a pin's status
Parameter 'literal': The pin's name, or literal
returns:

setPin(inindex:Integer state:Boolean)
Summary: Sets the pin indicated by the index to the desired state, which is either true or false
Parameter 'inindex': The index to the pin
Parameter 'state': The state to set the pin to
returns:

setPin(literal:String state:Boolean)
Summary: Sets the pin indicated by the used literal to the desired state
Parameter 'literal': The literal to be used
Parameter 'state':
returns:

setLiteralTranslation(lit:String pin:Integer)
Summary: Sets up a literal translator for a pin. Often names are easier to handle than Pin numbers. A feeder controled by pin 12 can be easier handled setLiteralTranslation("feeder",12) is used, so you can access pin 12 by setPin("feeder",true) instead of setPin(12,true) Pins can by design only be assigned to a single literal. Assigning a pin a second literal will overwrite the previous assignment. Literals need to be unique. Assigning the same literal to another pin will revoke the previous assignment.
Parameter 'lit': The literal you want the pin to be accessable under
Parameter 'pin': The pin to create a translation for
returns:

reportTranslation()
Summary: Reports the pin translation table, for debugging purposes The table is returned as literal:pin and is comma separated
returns: Reports the list of translated pins

maskPin(pin:Integer state:Boolean)
Summary: Enabled or disables the masking for a pin
Parameter 'pin': The pin to be masked
Parameter 'state': True will make the plugin ignore all state changes for this pin, false will allow state changes to be reported
returns:

maskPin(literal:String state:Boolean)
Summary: Enabled or disables the masking for a pin
Parameter 'literal': The literal of the pin to be masked
Parameter 'state': True will make the plugin ignore all state changes for this pin, false will allow state changes to be reported
returns:

makeOutput(pin:Integer state:Boolean)
Summary: Defines a pin as an output pin, or revokes output status

Parameter 'pin': The pin to be set as an output pin
Parameter 'state': If state is set to true, the pin will be defined an output pin, if its set to false the output status is revoked
returns:

makeOutput(pin:String state:Boolean)
Summary: Defines a pin as an output pin, or revokes output status
Parameter 'pin': The pin to be set as an output pin
Parameter 'state': If state is set to true, the pin will be defined an output pin, if its set to false the output status is revoked
returns:

lowlevel_write(port:Integer data:Array)
Summary: Writes data to the IOWarrior device
Parameter 'port': The port to write to, please refer to IOWarrior documentation
Parameter 'data': The data bytes to send
returns: Returns the number of written bytes

Events

litPinChanged
*description:*fires when a single pin has changed. Parameters: changed pin name, state of the pin.
*parameters:*OString OBoolean
pinsChanged
*description:*fires when pins have changed. All current pin states are reported as bitmap.
*parameters:*OInteger
pinChanged
*description:*fires when a single pin has changed. Parameters: changed pin no., state of the pin.
*parameters:*OInteger OBoolean

Plugin PlgKernScale Version: (1.0)

This plugin allows to utilize a KERN Scale Device via Serial Port.

Methods

open()
Summary: Opens the comport and initializes the handshaking with the scale device
returns:

getStableWeight()
Summary: Returns the Weight from the Scale, once the status is determined as "stable" (weight stayed constant for a given amount of time)
returns:

getWeight()
Summary: Instantly returns the current weight of the scale
returns:

tare()
Summary: Tare
returns:

setComDriver(comDriver:Plugin)
Summary: Connect this plugin with its communication driver for door operation and movement detection, For example a comport plugin.
Parameter 'comPlg': The comport plugin
returns:

Plugin PlgMouseGateSim Version: (1.0)

The mousegate simulator, simulates a virtual mousegate for debugging purposes.

Methods

setCanDriver(comPlg:Plugin)
Summary: Connect this plugin with its communication driver for door operation and movement detection, For example a comport plugin. The gate Driver will be ignored, this method is there for compatability purpose only
Parameter 'comPlg': The comport plugin
returns:

start()
Summary: Opens the neccessary connections to all of the plugins drivers.
returns:

showDisplay()
Summary: Brings up the gates gui, to visualize its status
returns:

hideDisplay()
Summary: Brings up the gates gui, to visualize its status
returns:

openDoor(no:Integer)
Summary: Opens a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

closeDoor(no:Integer)
Summary: Closes a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

triggerValve(no:Integer duration:Integer)
Summary: Activates the indicated vavle
Parameter 'no': Number of the Valve 1-3

returns:

askLeftPresence()
Summary: Checks if the right presence detector detects a mouse
returns:

askRightPresence()
Summary: Checks if the left mouse presence detector detects a mouse
returns:

checkDirectionBit(idx:Integer)
Summary: Allows to check a specific direction bit from the script
Parameter 'idx': The index to the bit 1-7
returns: Returns true if the bit is set, false if the the bit is not set

getWeight()
Summary: Reports the simulated weighting device's weight
returns:

setRFIDToDatamars()
Summary: Sets the RFID detector in datamars mode
returns:

setRFIDToTrovan()
Summary: Sets the RFID detector in datamars mode
returns:

setCanAddress(addr:Integer)
Summary: Sets the Adress of the RFID device to be used, default value will be 0 which should usually work for the mousegate
Parameter 'addr': The address to set
returns:

hideDisplay()
Summary: Brings up the gates gui, to visualize its status
returns:

setStateChecks(state:Boolean)
Summary: If this is set to true, error messages will be created if states conflict with a given command. (A closed door may not be closed) If its set to false, all states will be ignored.
Parameter 'state': The statechecks
returns:

openDoor(no:Integer)
Summary: Opens a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

closeDoor(no:Integer)
Summary: Closes a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

reCloseDoor(no:Integer)
Summary: Re-Closes a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

triggerValve(no:Integer duration:Integer)
Summary: Activates the indicated vavle
Parameter 'no': Number of the Valve 1-3
returns:

getDoorState(no:Integer)
Summary: Returns the state of the specified door.
Parameter 'no': Allowed range is 1-3
returns:

isDoorOpen(no:Integer)
Summary: Returns true if the specified door is marked open
Parameter 'no': Index of the door to be checked
returns:

isDoorClosed(no:Integer)
Summary: Returns true if the specified door is marked closed
Parameter 'no': Index of the door to be checked
returns:

setTare()
Summary: Sets the tare in the MouseGate's inbuild weighting device
returns:

constDoorOpen()
Summary: For safer comparisons, the constant for the states open, close and notfree can be quered via these methods.
returns:

constDoorClosed()
Summary: For safer comparisons, the constant for the states open, close and notfree can be quered via these methods.
returns:

constDoorBetween()
Summary: For safer comparisons, the constant for the states open, close and notfree can be quered via these methods.
returns:

constDirLeft()
Summary: For safer comparisons the constant for the direction detection can be queried
returns:

constDirRight()
Summary: For safer comparisons the constant for the direction detection can be queried
returns:

Events

doorState
*description:*fires when a door State uis changed. Parameters: door ID, state (1 = open, 2 = closed, 3 = udef)
*parameters:*OInteger OInteger
rfidDetect
*description:*fires when an Rfid ID is detected. Parameters: The id as byte array.
*parameters:*OArray
protocolFailure
*description:*fires when a protocol error occurs. Parameters: Error Message
*parameters:*OString
dirDetected
*description:*fires when a movement is detected. Parameters: direction (1 = left, 2 = right)
*parameters:*OInteger

Plugin PlgMouseGate Version: (1.0)

The mousegate simulator, simulates a virtual mousegate for debugging purposes.

Methods

setCanDriver(canDriver:Plugin)
Summary: Connect this plugin with its communication driver for door operation and movement detection, For example a comport plugin.
Parameter 'comPlg': The comport plugin
returns:

start()
Summary: Opens the neccessary connections to all of the plugins drivers.
returns:

showDisplay()
Summary: Brings up the gates gui, to visualize its status
returns:

hideDisplay()
Summary: Brings up the gates gui, to visualize its status
returns:

closeDoor(no:Integer)
Summary: Closes a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

closeDoor(no:Integer waitForStateChange:Boolean)
Summary: Closes a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
Parameter 'waitForStateChange': If this is set to false, the command will not wait for the door to finish opening
returns:

openDoor(no:Integer)
Summary: Opens a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

openDoor(no:Integer waitForStateChange:Boolean)
Summary: Opens a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
Parameter 'waitForStateChange': If this is set to false, the command will not wait for the door to finish opening
returns:

triggerValve(no:Integer duration:Integer)
Summary: Activates the indicated vavle
Parameter 'no': Number of the Valve 1-3
Parameter 'duration': Duration of the puff, specified in miliseconds 0-65535
returns:

askLeftPresence()
Summary: Checks if the right presence detector detects a mouse
returns:

askRightPresence()
Summary: Checks if the left mouse presence detector detects a mouse
returns:

checkDirectionBit(idx:Integer)
Summary: Allows to check a specific bit for the direction sensor.
Parameter 'idx': Index of the directionsensors bit 1-7
returns:

setTare()
Summary: Sets the TARE for the Mousegate's inbuild weighting device
returns:

setRFIDToDatamars()

Summary: Sets the RFID detector in datamars mode
returns:

setRFIDToTrovan()
Summary: Sets the RFID detector in datamars mode
returns:

setCanDriver(canPlg:Plugin)
Summary: Connect this plugin with its communication driver for door operation and movement detection, For example a comport plugin.
Parameter 'comPlg': The comport plugin
returns:

setCanAddress(addr:Integer)
Summary: Sets the Adress of the RFID device to be used, default value will be 0 which should usually work for the mousegate
Parameter 'addr': The address to set
returns:

showDisplay()
Summary: Brings up the gates gui, to visualize its status
returns:

hideDisplay()
Summary: Brings up the gates gui, to visualize its status
returns:

setStateChecks(state:Boolean)
Summary: If this is set to true, error messages will be created if states conflict with a given command. (A closed door may not be closed) If its set to false, all states will be ignored.
Parameter 'state': The statechecks
returns:

openDoor(no:Integer)
Summary: Opens a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

closeDoor(no:Integer)
Summary: Closes a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

reCloseDoor(no:Integer)
Summary: Re-Closes a door on the MouseGate
Parameter 'no': Number of the door to be opened, can be 1-3
returns:

triggerValve(no:Integer duration:Integer)
Summary: Activates the indicated vavle
Parameter 'no': Number of the Valve 1-3
returns:

getDoorState(no:Integer)
Summary: Returns the state of the specified door.
Parameter 'no': Allowed range is 1-3
returns:

isDoorOpen(no:Integer)
Summary: Returns true if the specified door is marked open
Parameter 'no': Index of the door to be checked
returns:

isDoorClosed(no:Integer)
Summary: Returns true if the specified door is marked closed
Parameter 'no': Index of the door to be checked
returns:

getWeight()
Summary: Reports the weight, reported from the scale
returns:

constDoorOpen()
Summary: For safer comparisons, the constant for the states open, close and notfree can be quered via these methods.
returns:

constDoorClosed()
Summary: For safer comparisons, the constant for the states open, close and notfree can be quered via these methods.
returns:

constDoorBetween()
Summary: For safer comparisons, the constant for the states open, close and notfree can be quered via these methods.
returns:

constDirLeft()
Summary: For safer comparisons the constant for the direction detection can be queried
returns:

constDirRight()
Summary: For safer comparisons the constant for the direction detection can be queried
returns:

Events

doorState
*description:*fires when a door State uis changed. Parameters: door ID, state (1 = open, 2 = closed, 3 = udef)
*parameters:*OInteger OInteger
rfidDetect
*description:*fires when an Rfid ID is detected. Parameters: The id as byte array.
*parameters:*OArray
protocolFailure
*description:*fires when a protocol error occurs. Parameters: Error Message
*parameters:*OString
dirDetected
*description:*fires when a movement is detected. Parameters: direction (1 = left, 2 = right)
*parameters:*OInteger

Plugin PlgRFIDreader Version: (1.0)

This plugin allows to utilize the RFIDreadermodule via Serial Port.

Methods

open()
Summary: Opens the comport and initializes the handshaking with the scale device
returns:

getRFID()
Summary: Returns the RFID from RFID scanning buffer
returns:

getRFIDstream()
Summary: Initiates constant stream of current RFID readings
returns:

shutdownDisable()
Summary: Disable shutdown (reader disables after 30s)
returns:

setComDriver(comDriver:Plugin)
Summary: Connect this plugin with its communication driver for door operation and movement detection, For example a comport plugin.
Parameter 'comPlg': The comport plugin
returns:

Plugin PlgWeightBox Version: (1.0)

This plugin allows to utilize the RFIDreadermodule via Serial Port. Info for all commands that end in "Devices", a bitmap is specified as followed: 00000000 00000000 00000000 00000000 | | dev 25-32 dev 17-24 dev 9-16 dev 1-8 The value is then passed as a decimal value

Methods

open()
Summary: Opens the comport and initializes the handshaking with the scale device
returns:

requestDataAllDevices()
Summary: Sends a get data request to all devices
returns:

requestDataExtendedAllDevices()
Summary: Sends a get data request to all devices, the "extended" protocol is used to report the data
returns:

instantRequestDataAllDevices()
Summary: Instructs all connected devices to report data instantly. Does not work when more than one device is connected.
returns:

requestDataFromDevice(devid:Integer)
Summary: Sends a get data request to a specific device
Parameter 'devid': The device to send the command to (1-32)
returns:

clearAllDevices()
Summary: Sends a clear request to all devices
returns:

clearDevice(devidx:Integer)
Summary: Instructs a specific device to clear
Parameter 'devidx': The device's index id.
returns:

clearDevices(map:Integer)
Summary: Instructs specific devices to clear
Parameter 'map': A bitmap of all devices to clear
returns:

taraOnDevice(devidx:Integer)
Summary: Instructs a specific device to do a tara
Parameter 'devidx': The device's index id.
returns:

taraOnDevices(map:Integer)
Summary: Instructs specified devices to do a tara
Parameter 'map': The devices to do tara as a bitmap
returns:

taraAllDevices()

Summary: All connected devices are instructed to do a tara command
returns:

logModeAllDevices()
Summary: All connected devices are instructed to go into logmode
returns:

resetDevice(devid:Integer)
Summary: The specified device deletes its data
Parameter 'devid': The device to do a reset
returns:

resetDevices(map:Integer)
Summary: The specified devices reset their data
Parameter 'map': The devices to do a reset, specified by bitmap
returns:

resetAllDevices()
Summary: All connected devices are instructed to go into logmode
returns:

setAverageRateForAllDevices(rate:Integer)
Summary: All connected devices are instructed to go into logmode
Parameter 'rate': The average rate to be used by all devices, can be (6-120)
returns:

getStatusInfoFromDevice(devid:Integer)
Summary: Returns the status info for the specified device
Parameter 'devid': The device to report status info
returns:

getStatusInfoFromDevices(map:Integer)
Summary: Returns the status info for the specified devices (by bitmap)
Parameter 'map': Bitmap
returns:

getWeightFromDevice(devid:Integer weightIDX:Integer)
Summary: Requests the current weight from a device
Parameter 'devid': The id to request the current weight from
Parameter 'weightIDX': The weight to return, can be 1 or 2
returns: Returns the weight as double value

setWeightChangeDelta(delta:Double)
Summary: Sets the Delta between the last Reported Weight, and the Current weight that needs to be exceeded for the WeightChange event to be triggered. Please note that lower values might mean a constant triggering of the event.
Parameter 'delta': The delta, default is 0.1
returns:

pollWeightChange(devid:Integer)
Summary: This basically creates a getWeight command fot the specified device that will not raise a changeWeight event unless the weight exceeds the specified delta Basically identical to RequestDataFromDevice, but here as a reminder for the weightChangeProcedure
returns:

setComDriver(comDriver:Plugin)
Summary: Connect this plugin with its communication driver for door operation and movement detection, For example a comport plugin.
Parameter 'comPlg': The comport plugin
returns:

Events

dataReceived
*description:*fires when data is received. Parameters: Device ID, Weight1, Weight2.
*parameters:*OInteger ODouble ODouble
weightChanged
*description:*fires when the weight has changed beyond delta allowed by .setWeightChangeDelta. Parameters: Device ID, Weight ID, Weight
*parameters:*OInteger OInteger ODouble