Table of contents

< Back
Table of contents
Forward >
1. Introduction

1. Introduction

Important notices!

- The DestinyPatch has been developed by Bananen-Joe. It is an extension for the RPG-Maker 2000, which has been developed by the company Enterbrain. This extension is <u>not</u> an official extension from Enterbrain.
- During the development of the patch the author acted with best knowledge and conscience, however errors are not excludable. Hence the author dissociates himself from every direct or indirect damage that could occur. Though the author is grateful for each helpful hint, which can be used to correct errors.
- The entire DestinyPatch is a free software. Each distribution with costs is illegal unless the author permits this emphatically.
- This help file refers often to external pages (e. g. like Wikipedia). The author dissociates himself from every external content, because the internet is a incessantly changing medium (e. g. everyone can edit Wikipedia articles). During development of this help file all external links were correct and assists the content of this help file with explanations, etc. Hence it is not possible in this static help file to react on external contents that has been changed.

Embedding the patch

A detailed manual how to embed the Destiny.dll into a RPG-Maker 2000 project is included in the help file for the DestinyPatcher. The interrelationship of the programs is explained there, too. In this help file here (the one you are currently reading) is only a manual for the scripting language (DestinyScript) included!

The layout of this help file

During development of this help file some formats have been designed and abided. These should help the user for faster navigating and getting a better overview.

Headline

The headline is written on the upper border of each page.

The footer is on the lower border of each page. On the left of this footer is the point "< Back", which can be used to navigate to the previous page. Under this point is the headline of the previous point written. On the right of this footer is the point "Forward >", which can be used to navigate to the following page. Under this point is the headline of the previous point written. In the center of the footer is the headline of the current page written again.

Internal links inside of this help file are colored blue and not underlined unless you hover over them with the mouse cursor.

External links are colored green and always underlined. If you click on an external link then it will open a new window.

```
$
Variable = Function("String", 12345);
```

In code examples the script code is written in one box. On the left border are line numbers. The following explanations refer to them. The code itself is colored multiple times to make the terms much easier to read in refer to their meaning.

- Variables & functions are colored blue
- Symbols are colored orange
- Numbers are colored purple
- Strings are colored red

Information
This is a short additional piece of information, which can help you to avoid some problems.

Important additional information are written in yellow boxes.

---

2. Assembly of the scripting language

Calling a DestinyScript

The comment function of the RPG-Maker is used to call a DestinyScript. Because they are not only used for DestinyScripts (they are used for "common" comments, too) every DestinyScript begins with a dollar sign ($). This symbol has been chosen because it looks like a S and the word "script" starts with a s. It is important that the <u>first</u> char of the RPG-Maker comments needs to be the $ sign (this even means there may not be spaces or something else before the dollar sign) otherwise the RPG-Maker comment will not be interpreted as DestinyScript.

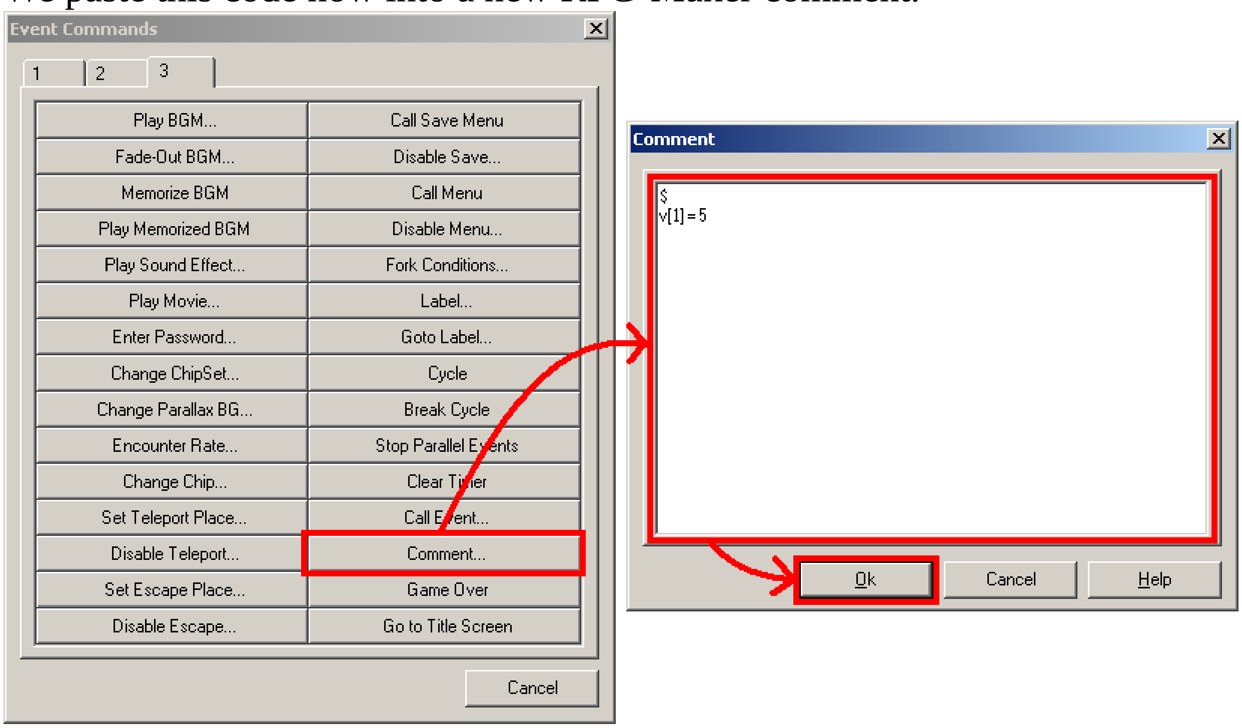
> ***Information****If a RPG-Maker comment should be a "common" comment, but starts with a $ sign, it will be interpreted as DestinyScript and could show error messages. You can solve the problem if you put a space in the front of the $ sign. So the $ sign isn't the first symbol of the RPG-Maker comment anymore. In the RPG-Maker event command list the comment command looks like nearly the same.*

The first example

At first we create a new (empty!) RPG-Maker project. Next we modify the RPG_RT with the RPG_RT.exe so that it loads the Destiny.dll. (Version: 1.0, Language: English, MessageLink: activiated, Number of dwords: 100, Number of doubles: 100, Number of strings: 100)
After generating the project we create a new event. First of all we have this code:

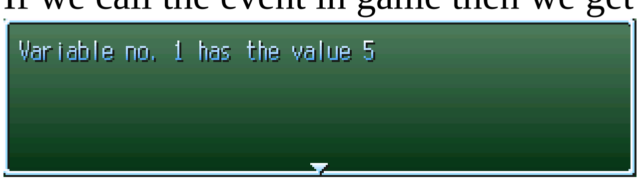
```
$
v[1] = 5
```

We paste this code now into a new RPG-Maker comment.



This graphic can vary from the used RPG-Maker version.

We insert a MessageBox among that comment, which displays the content of the first variable. (e. g. "Variable no. 1 has the value \v[1]")
If we call the event in game then we get the following result:



We can see that variable no. 1 has the value 5. But why that? All variables of the RPG-Maker are by default initialized to 0. And given that it is a new (and complete empty) project there must something else been happen. The answer is obvious: The DestinyScript assigned the value 5 to the variable no. 1!

Assembly of a DestinyScript

If we take a look at the example code again and think about its result then the meaning of the script will be clear.

```
1 $
2 v[1] = 5
```

In line no. 1 is the $ sign which tells the Destiny.dll that the RPG-Maker comment should be interpreted as DestinyScript.
In line no. 2 is the command which assigns the value 5 to variable no. 1. Such a command has always the same assembly. On the left side is the destination of the operation. In this case it is variable no. 1 (hence the v[1] - v stands for variable and [1] for the index!). In the center is the used operator. In this example this is the equal sign (=). On the right side is that what we want to assign to the destination (= source). In this case it is the number 5.
A DestinyScript may contain multiple commands. To separate them a semicolon (;) is used. A command needn't to be in a single line. But it is possible to write the entire DestinyScript into a single line (the leading $ sign needs not its own line!).

Information
The last command of a DestinyScript may end with a semicolon but it needn't to end with a semicolon.

As example a DestinyScript with more than one command could look like this:

```
$
v[1] = 5;
v[2] = 23
```

If we execute this DestinyScript the variable no. 1 will be 5 and variable no. 2 will be 23.

Information
Multiple commands can be written inside of a RPG-Maker comment. But the leading $ sign is only required once. Hence the $ sign may not be in front of each command!

It is expedient to make a important note here. If an error occurs during execution of a DestinyScript then the entire script will be aborted. This can be beneficial or even not. To avoid this you could write each command into a single RPG-Maker comment or change the error handling with the Errors object.

3. Data types

What are data types?

As a user of the RPG-Maker 2000 you already know 2 data types. At first variables and as second switches. But you can't compare a variable and a switch directly, because they save complete different values. A RPG-Maker 2000 variable can save values in the range from -999999 to +999999 (in the Destiny.dll you have a greater range - see table). But you can only use integer values (e. g. floating point numbers like 1.5 won't work!). A switch however can save only one of two values: on or off (constants: True and False). These two data types are available in different scopes (one for variables and one for switches). The Destiny.dll extends the RPG-Maker game by 3 additional scopes. At first a scope of the data type dword is added. This data type is complete identical with the data type of the variable. At second a scope of the data type double is added. This data type can save floating point numbers with a very huge range. At third a scope with the data type string is added. This data type saves text instead of numbers (e. g. names, words or complete sentences).

List of data types

The DestinyPatch knows 7 different data types.

Data type	Size	Range	Usage
Variable	4 bytes	-2147483648 ... +2147483647	*This data type is equivalent to the data type "dword"*
Switch	1 byte	0 ... +1	This data type can save only boolean values (yes or no). Usually the boolean constants True (= 1) and False (= 0) are used for this data type.
Dword	4 bytes	-2147483648 ... +2147483647	This is the most common data type. It saves only integer values.
Word	2 bytes	-32768 ... +32767	This data type is similar to dword, but this one has a smaller range. This data type is used only by a few methods, but its usable for data transmission. You can save traffic with it, because it needs only two bytes. This data type saves only integer values, too.
Byte	1 byte	0 ... +255	This data type is similar to dword, but this one has a constitutive smaller range. This data type saves only very small numbers and needs hence only one byte in memory. This data type is used for pixels in a picture. This data type saves only integer values, too.
Double	8 bytes	-1.7E+308 ... -5.0E-324 ... +5.0E-324 ... +1.7E+308	This data type can save very huge numbers, but in favor it requires much more space in memory. The accurancy is 15 integer/decimal places. This is the only data type that can save floating point numbers.
String	4 + n bytes	ca. 0 ... +2147483647	This data type saves text (e. g. names, words or complete sentences).

		Zeichen	

Scopes

To save the result of an operation it is necessary to have some scopes in memory where the result can be stored. Scopes for the data types variable and switch are allocated by the RPG_RT.exe. The other scopes (for the data types dword, double and string) are allocated by the Destiny.dll. To access the scopes you can use their abbreviations.

Abbreviation	Data type	Data source	Identifier
v[ ]	Variable	RPG_RT.exe	V stands for variable.
s[ ]	Switch	RPG_RT.exe	S stands for switch.
d[ ]	Dword	Destiny.dll	D stands for dword.
f[ ]	Double	Destiny.dll	F stands for floating point number.
a[ ]	String	Destiny.dll	A stands for ANSI string.

We have already seen how to access a variable via the v abbreviation. You simply write down the abbreviation with some brackets behind it (the brackets must be written directly after the abbreviation!). You write down the index of the element you want to access inside the brackets (e. g. v[1] for the first variable, v[2] for the second variable, ...). The same principle works with other scopes, too.

```
$
d[5] = 100;
v[3] = d[5]
```

If we paste this example into a RPG-Maker comment and let subsequent display the value of the variable no. 3 in a MessageBox then we can see that the variable has the value 100. In this example we assigned first the value 100 to the dword no. 5. Subsequently we assigned the value of the dword no. 5 to the variable no. 3. On this way we assigned a value from a different scope to the variable no. 3.

Indirectly addressing

So far we accessed the elements of a scope only directly (e. g. v[1] for the first variable). An other way to access an element of a scope is the indirectly addressing. To do this we simply write an element of a scope instead of an immediate (fixed) number into the brackets.

```
1 $
2 v[4] = 5;
3 v[v[4]] = 17
```

If we execute this script and take a subsequently a look at variable no. 5 then we can see that its value is 17. In this example we accessed variable no. 5 indirectly. First we assigned in line 2 the value 5 (the index of the variable we want to access) to the variable no. 4. Subsequently we wrote in line 3 that we want to use the value of variable no. 4 as index instead of an immediate number.

Information
All scopes can be addressed indirectly. The scopes used for indirectly addressing can be addressed indirectly, too (e. g. v[v[v[1]]]). The maximum depth of indirectly addressing depends on the capacity of the computer where the game is running. Usually you needn't a deeper addressing depth than 1.

Conversion of data types

If the data types differ (e. g. if you want to assign a dword to a double) then the data types will be converted automatically. This happens inside of formulas or parameters, too. Only the mixed calculation of strings and numbers or switches and numbers raise an error. To do a calculation like this you must explictly convert the values using the Convert object.

< Back		**Forward >**
2. Assembly of the scripting language	3. Data types	4. Number formats

4. Number formats

What are number formats?

Number formats are different ways to write down a number in DestinyScript. There are 4 formats total to write down a number in DestinyScript. Three of them are known from computer science. The other format is used for our known decimal system. If we write down a number then we usually use one of the 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Because we have 10 digits the base of this number system is 10. If we read a number then we multiply each digit with the base exponentiated by the digit index minus 1.

In this example the circumflex is used as symbol for the exponentation.

```
123 = 1 * 10 ^ 2 + 2 * 10 ^ 1 + 3 * 10 ^ 0
123 = 1 * 100 + 2 * 10 + 3 * 1
123 = 100 + 20 + 3
123 = 123
```

A translation of a number from the decimal system into the decimal system meaningless, because the result will always be the same. However if we use two different number systems (number systems with different bases) then we get differing results. First we use the hexadecimal system. It contains 16 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). The digits A to F equates to the numbers 10 to 15. For example: 1AB is a valid hexadecimal number. Translated into the decimal system it is:

In this example the circumflex is used as symbol for the exponentation.

```
1AB (hex) = 1 * 16 ^ 2 + 10 * 16 ^ 1 + 11 * 16 ^
0 (dec)
1AB (hex) = 1 * 256 + 10 * 16 + 11 * 1 (dec)
1AB (hex) = 256 + 160 + 11 (dec)
1AB (hex) = 427 (dec)
```

It seems to be meaningless for the layman, but for the expert it is very helpful. If we write down such numbers we find a drastic problem: Some numbers can't be clearly related to a specific number system. For example the number 10 in hexadecimal is not the same as the number 10 in decimal. Hence we must cleary define which number system we use. So we append a 0x as identifier in DestinyScript at the beginning of a hexadecimal number (e. g. 0x123 = 291). Something similar is used for the other number formats octal (Base: 8, Digits: 0-7, Identifier: 0o) and binary (Base: 2, Digits: 0 and 1, Identifier: 0b). For decimal numbers are no identifiers used. An important note is still that floating point numbers can only be used in the decimal number format. The point is used as decimal separator (e. g. 1.5 is a valid floating point number).

List of number formats

DestinyScript supports 4 different number formats

Number format	Base	Floating point number	Identifier	Examples
Decimal	10	yes	*none*	11, 9, 10.6, 3.1415926535
Hexadecimal	16	no	0x	0x10, 0xAC, 0x1237
Octal	8	no	0o	0o17, 0o32, 0o700
Binary	2	no	0b	0b111011, 0b00111101, 0b1111110

Information
If we try to write down a floating point number in a different number format than the decimal system then an error occurs (e. g. 0x11.5 isn't a valid number).

Differently calculation of integers and floating point numbers

First of all we have the following example:

```
1 $
2 f[1] = 3 / 2;
3 f[2] = 3.0 / 2.0
```

You could anticipate that f[1] and f[2] are equal at the end of the script, but in fact they are different: f[1] whould be 1 and f[2] whould be 1.5 at the end of the script. This difference occurs because we defined each number in line 2 as integer values. So they are divided like integers (integers don't have decimal places - hence they are not calculated). In line 3 instead we defined each number as floating point number. Hence they are calculated like floating point numbers (the decimal places are not lost). You should notice this difference if you want to calculate with floating point numbers.

5. Strings

Defining strings

Strings are not numbers. However its content must be able to declare in DestinyScript without creating mistakes (e. g. a string could contain numbers which should not be interpreted like numbers). The solution is to write down the content of the string into quotes (").

```
$
a[1] = "Content of a string"
```

Information
If you want to use quotes inside of a string then you must use the QUOTE constant (e. g. a[1] = "Hello with " + QUOTE + "quotes" + QUOTE).

Concatentation of strings

If you want to concatenate some strings you must use the add operator. A more specific description can be found at 6. Operators.

Strings with multiple lines

If a string should contain multiple lines then you must use the CRLF constant.

```
$
a[1] = "Line 1" + CRLF + "Line 2"
```

If you display this string into a MessageBox you will have some trouble, because the MessageBox of the RPG_RT processes lines on an other way than the Destiny.dll. A more specific description can be found at the string placeholder of the MessageLink.

< Back 4. Number formats | 5. Strings | **Forward >** 6. Operators

6. Operators

What are operators?

Operators are signs that represent specific arithmetical or binary operations. DestinyScript supports 9 operators. The set operator is neutral, because it can be used with each data type. Furthermore there are 5 arithmetical operators (addition, subtraction, multiplication, Division and Modulo). The addition ist something special, because it can be used for the concatenation of strings, too. At last there are 3 binary operators (AND, OR and XOR).

Set

Description

The simpliest operator is the set operator we already know. With this operator it is possible to assign a value to a destination directly. This operator can only be used at the beginning of a command. This operator works with each data type. The equality sign (=) is used for this operator.

Signs

Signs	Beginning?
=	yes

Example

```
$
v[1] = 1
```

Addition

Description

You can add two values with the addition operator. This operator can be used at the beginning and in the rear part of a command. If you apply an addition at the beginning of a command the rear part of the command will be processed as if it is written in parentheses. If this operator is used with strings then the values will be concatenated. Switches can't be added. The plus sign (+) is used for this operator.

Signs

Signs	Beginning?
+	no
+=	yes

Example

```
$
v[1] = 1 + 10 + 100;
v[1] += 20;
a[1] = "Joe";
a[2] = "Hello " + a[1]
```

The variable no. 1 whould be 131 at the end and the string no. 2 whould be "Hello Joe".

Information
If you use the addition operator to concatenate strings then you may only use strings. For example this command whould raise an error:
a[1] = "Number equals " + v[1]
Instead all data types must be converted to strings first if you concatenate them. This command whould be valid:
a[1] = "Number equals " + Convert.String(v[1])
The Convert object is used here.

Subtraction

Example

You can subtract two values with the subtraction operator. This operator can be used at the beginning and in the rear part of a command. If you apply a subtraction at the beginning of a command the rear part of the command will be processed as if it is written in parentheses. Strings and switches can't be subtracted. The minus sign (-) is used for this operator.

Signs

Signs	Beginning?
-	no
-=	yes

Example

```
$
v[1] = 50 - 7;
v[1] -= 9
```

The variable no. 1 whould be 34 at the end.

Multiplication

Description

You can multiply two values with the multiplication operator. This operator can be used at the beginning and in the rear part of a command. If you apply a multiplication at the beginning of a command the rear part of the command will be processed as if it is written in parentheses. Strings and switches can't be multiplied. The multiplication sign (*) is used for this operator.

Signs

Signs	Beginning?
*	no
*=	yes

Example

```
$
v[1] = 7 * 3;
v[1] *= 3
```

The variable no. 1 whould be 63 at the end.

Division

Description

You can divide two values with the division operator. This operator can be used at the beginning and in the rear part of a command. If you apply a division at the beginning of a command the rear part of the command will be processed as if it is written in parentheses. Strings and switches can't be divided. If you divide through zero an error will occur. The division slash (/) is used for this operator.

Signs

Signs	Beginning?
/	no
/=	yes

Example

```
$
v[1] = 100 / 10;
v[1] /= 2
```

The variable no. 1 whould be 5 at the end.

Modulo

Description

In the modulo operation the left value will be divided by the right value (integer division) and the remainder will be returned (this is the part of the left number, which whould be required to build the decimal part of the result). This operation can be used at the beginning and in the rear part of a command. If you apply a modulo operation at the beginning of a command the rear part of the command will be processed as if it is written in parentheses. Strings and switches can't be divided. If you divide through zero an error will occur. The percent sign (%) is used for this operation.

Signs

Signs	Beginning?
%	no
%=	yes

Example

```
$
v[1] = 100 % 3;
v[2] = 99;
v[2] %= 10
```

The variable no. 1 whould be 1 at the end and variable no. 2 whould be 9.

AND operator

Description

You can apply a binary AND operation with two values via the AND operator (this means that every bit is AND operated). This operation can be used at the beginning and in the rear part of a command. If you apply an AND operation at the beginning of a command the rear part of the command will be processed as if it is written in parentheses. Strings, switches and doubles can't be AND operated. The ampersand sign (&) is used for this operation.

Sings

Sings	Beginning?
&	no
&=	yes

Example

```
$
v[1] = 100 & 7;
v[2] = 31;
v[2] &= 255
```

The variable no. 1 whould be 4 at the end and variable no. 2 whould be 31.

OR operation

Description

You can apply a binary OR operation with two values via the OR operator (this means that every bit is OR operated). This operation can be used at the beginning and in the rear part of a command. If you apply an OR operation at the beginning of a command the rear part of the command will be processed as if it is written in parentheses. Strings, switches and doubles can't be OR operated. The pipe sign (|) is used for this operation.

Signs

<table>
<tr><th>Signs</th><th>Beginning?</th></tr>
<tr><td>|</td><td>no</td></tr>
<tr><td>|=</td><td>yes</td></tr>
</table>

Example

```
$
v[1] = 7 | 3;
v[2] = 1;
v[2] |= 100
```

The variable no. 1 whould be 7 at the end and variable no. 2 whould be 101.

XOR operation

Description

You can apply a binary EXCLUSIVE OR operation (XOR) with two values via the XOR operator (this means that every bit is XOR operated). This operation can be used at the beginning and in the rear part of a command. If you apply a XOR operation at the beginning of a command the rear part of the command will be processed as if it is written in parentheses. Strings, switches and doubles can't be XOR operated. The circumflex sign (^) is used for this operation.

Signs

Signs	Beginning?
^	no
^=	yes

Example

```
$
v[1] = 27 ^ 13;
v[2] = 1;
v[2] ^= 3
```

The variable no. 1 whould be 22 at the end and variable no. 2 whould be 2.

7. Leading signs

Description

There are three leading signs in DestinyScript totally: two arithmetical (Plus and Minus) as soon as a binary (NOT). Please notice that every sign must be directly at the beginning of each values/scopes/parentheses. The data types switch and string cannot be used with leading signs.

Plus sign

Description

The plus sign has only been introduced to make it easier to copy numbers. It doesn't change the number. The plus sign (+) is used for this leading sign.

Sign
+

Example

```
$
v[1] = 0 + +1
```

The variable no. 1 whould be 1 at the end.

Minus sign

Description

You can define negative numbers by using the minus sign (if you use an integer value the two's complement will be formed). The minus sign (-) is used for this leading sign.

Sign
-

Example

```
$
v[1] = 5 + -3;
v[2] = 10 - -11
```

The variable no. 1 whould be 2 at the end and variable no. 2 whould be 21.

Information
The mathematical law "minus minus is plus" is used here, too.

NOT sign

Description

You can negate a value with the NOT sign (the ones' complement will be formed). The tilde sign (~) is used for this leading sign. This leading sign can't be used with doubles.

Sign
~

Example

```
$
v[1] = ~100
```

The variable no. 1 whould be -101 at the end.

Information
According to the boolean algebra is not not the same as the value without any not sign before it (= Identity)!

8. Formulas

Formulas (generic)

After we know the data types, number formats, strings, operators and leading signs we know (nearly) all what's required to use formulas in DestinyScript. The last missing parts are parentheses and priorities of the operators.

Priorities

Description

According to the arithmetical algebra it is necessary: point operations (multiplication, division) will be executed before stroke operations (addition, subtraction). And according to the boolean algebra is necessary: AND will be executed before OR. A combination of these laws forms the priorities of the operators. If a formula contain multiple partial calculations then they are calculated in the order that result from their priorities. If more than one partial calculation have the same priority then they are calculated from left to right. Leading signs (plus, minus, NOT) have always the highest priority.

```
x = 15 + 3 * 9
x = 15 + 21
x = 36
```

In this example the point operation (3 * 9) has been calculated before the stroke operation (which whould be 15 + 3) was calculated.

List of the priorities

In the following list applies: the higher the number the higher the priority.

Operator	Priority
AND operation	4
OR operation	3
XOR operation	3
Modulo	2
Division	2
Multiplication	2
Subtraction	1
Addition	1

Example

The following example uses the signs of DestinyScript but it is just a calcuation example - it is not a valid DestinyScript.

```
x = 1000 + 2 - 10 * 3 / 15 % 8 | 3 ^ 7 & 6
x = 1002 - 10 * 3 / 15 % 8 | 3 ^ 7 & 6
x = 1002 - 30 / 15 % 8 | 3 ^ 7 & 6
x = 1002 - 2 % 8 | 3 ^ 7 & 6
x = 1002 - 2 % 11 ^ 7 & 6
x = 1002 - 2 % 11 ^ 6
x = 1002 - 2 % 13
x = 1002 - 2
x = 1000
```

In this example some lines could have been calculated in a single step, but to make it easier to understand the calculation has been made step-by-step.

Parentheses

Description

If a partial calculation with less priority should be calculated before a partial calculation with high priority then you must put the calculation with less priority in parentheses ( ).

Example

The following example uses the signs of DestinyScript but it is just a calcuation example - it is not a valid DestinyScript.

```
x = 100 * -(10 + 7)
x = 100 * -17
x = -1700
```

Information
A sign at the beginning of a pair of parentheses modifies only the result of the calculation in the parentheses.

9. Objects

Description

Objects are that what represents the DestinyPatch. All functions and properties of the RPG-Maker 2000 game (and even those of the Destiny.dll) are bunched into objects. These functions (further called methods) and properties can be accessed via the relevant object. For example you can access the keyboard using the Keyboard object. So you can use the object to check the key states. As already said an object can contain two types of content: methods and properties. Those content types are formatted differently. Methods are always write-protected, properties only sometimes (see the definition of the relative property).

Syntax

The syntax to call an object is always the same:

```
1 $
2 v[1] = Objectname.Functionname(Parameter1,
3 ...);
4 Objectname.Functionname(Parameter1, ...);
5 v[2] = Objectname.Propertyname;
  v[3] = Objectname[Index].Propertyname
```

In this example you see several ways to access the content of an object. One thing is always the same: first you write down the object name, then a dot and at last the function or property name. Between this identifiers you may not use any spaces.

If you call a method (Line 2) you must add a pair of parentheses which must be written directly after the method name (this means without spaces). Inside of these parentheses you write the parameters (if there are some). To separate the parameters you use the comma. If you call a method you must always write down the parentheses even if the method has no parameters!

The result of the method (= return value) can be used in a formula (e. g. the return value will be stored into variable no. 1 at line 2).

Sometimes a method has no return value (Line 3). In this case you can't access its return value in formulas.

If you access a property (Line 4) you do it like a method call (but without the parentheses and the parameters).

Some objects or properties have an index (Line 5). This index identifies which element of the object/property should be accessed. The index is written into a pair of brackets [ ]. Like the parentheses they are written directly after the object name/property name (this means without spaces). If an index has more than one dimension (e. g. a two dimensional array) then you separate the indices with a comma.

Information

If a data type in a parameter differs from the specified one (e. g. the data type string is required and the data type dword is used) then the used parameter is converted to the required data type automatically (e. g.

dword is converted into string). You needn't to call an extra conversion method. (Example: v[1] = String.Length(d[1]);)

Examples

```
1 $
2 v[1] = Keyboard.GetKey();
3 Keyboard.SetKeyState(VK_RIGHT,
4 KEYEVENTF_KEYDOWN);
5 v[2] = Time.Day;
6 v[3] = Picture[1].Width;
7 v[4] = Map.Lower[2, 3];
8 v[5] = 5 - Math.Abs(v[6]);
  Mouse.X = 10
```

In line 2 is a habitually method call (without parameters). The return value (in this case it is the last pressed key) will be stored into the variable no. 1.
In line 3 is a method without return value (but with parameters) called (in this case the key state will be set).
In line 4 a property is requested. The property value (in this case the number of the current day of the month) will be stored into variable no. 2.
In line 5 a property of an object is requested which requires an index. The property value (in this case the width of picture no. 1) will be stored into variable no. 3.
In line 6 the value of a property (which has a two dimensional index) of an object is requested. The property value (in this case number of the chip at position 2, 3) will be stored into variable no. 4.
In line 7 a method is called (with parameters) inside of a formula. The return value of the method (in this case the absolute value of variable no. 6) will be subtracted from the number 5 and then stored into variable no. 5.
In line 8 the value 10 will be assigned to the property of an object (in this case the x coordinate of the mouse cursor).

> ***Information***
> *The upper/lower case of object/method/property names are irrelevant. OBJECTNAME.PROPERTY, objectname.property or even oBjEcTnAmE.PrOpErTy cause all the same.*

Alphabetical list of objects

Objectname	Short description
Actor	List of heroes
Client	Connections via internet/network
Convert	converting of different data types
Destiny	Current options of the Destiny.dll
Directory	Listing directories and abstract file system operations
Error	Handling of single errors
Errors	Handling of all errors
Event	List of events (EventID)
File	Reading/writing files
Game	The current game
Keyboard	Key queries
Logic	Logical operations and comparisons
Map	Current map
MapEvent	List of events (serially numbered)
Math	Miscellaneous mathematical functions
Mouse	Cursor position
Picture	List of pictures
Server	Incoming connections via internet/network
String	Miscellaneous string functions
Time	Time queries

9.1 Destiny object

Description

The Destiny object represents the Destiny.dll and can be used to query the version of the Destiny.dll, setting up the language, as soon as saving/loading the scopes of Destiny.dll.

List of methods/properties

Name	Type	Short description
VersionMajor	Property	The integer part of the used Destiny.dll version
VersionMinor	Property	The decimal part of the used Destiny.dll version
DllVersionMajor	Property	The integer part of the available Destiny.dll version
DllVersionMinor	Property	The decimal part of the available Destiny.dll version
Language	Property	The used language of the Destiny.dll
Save	Method	Saves the values of all scopes of the Destiny.dll
Load	Method	Loads the values of all scopes of the Destiny.dll

VersionMajor

Description

Returns the integer part of the used Destiny.dll version. For example if a version 3.4 whould be available and the version 1.2 whould be used this property whould return 1.

Syntax

```
Destiny.VersionMajor
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Destiny.VersionMajor
```

On Destiny.dll version 1.0 v[1] whould be at end: 1

VersionMinor

Description

Returns the decimal part of the used Destiny.dll version. For example if a version 3.4 whould be available and the version 1.2 whould be used this property whould return 2.

Syntax

```
Destiny.VersionMinor
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Destiny.VersionMinor
```

On Destiny.dll version 1.0 v[1] whould be at end: 0

DllVersionMajor

Description

Returns the integer part of the available Destiny.dll version. For example if a version 3.4 whould be available and the version 1.2 whould be used this property whould return 3.

Syntax

```
Destiny.DllVersionMajor
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Destiny.DllVersionMajor
```

On Destiny.dll version 1.0 v[1] whould be at end: 1

DllVersionMinor

Description

Returns the decimal part of the available Destiny.dll version. For example if a version 3.4 whould be available and the version 1.2 whould be used this property whould return 4.

Syntax

```
Destiny.DllVersionMinor
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Destiny.DllVersionMinor
```

On Destiny.dll version 1.0 v[1] whould be at end: 0

Language

Description

This is the chosen language of the Destiny.dll. If you change this value you change the language of the error messages, too.

Syntax

```
Destiny.Language
```

Data type

Dword

Type

Property

Range

0: Language German
1: Language English

Example

```
$
Destiny.Language = 1
```

At the end all error messages whould be english.

Save

Description

Saves all scopes of the Destiny.dll (dwords, doubles and strings) into a file. The file will be stored into the game directory and has the name SaveXX.dsd (XX will be replaced with the slot in two digit format).

Syntax

```
Destiny.Save(Slot)
```

Return value

None

Type

Method

Parameter: Slot

Description

The number of the save slot.

Data type

Dword

Range

0 to 99

Example

```
$
Destiny.Save(1)
```

At the end all d[...], f[...] and a[...] scopes whould be saved in the file Save01.dsd.

Load

Description

Loads the scopes of the Destiny.dll (dwords, doubles and strings) from a file. The file is stored into the game directory and has the name SaveXX.dsd (XX will be replaced with the slot in two digit format).

Syntax

```
Destiny.Load(Slot)
```

Return value

None

Type

Method

Parameter: Slot

Description

The number of the save slot.

Data type

Dword

Range

0 to 99

Example

```
$
Destiny.Load(1)
```

At the end all d[...], f[...] and a[...] scopes whould be loaded from the file Save01.dsd.

9.2 Game object

Description

The Game object represents the RPG_RT.exe and can be used to save/load the game or even quit it.

List of methods/properties

Name	Type	Short description
Save	Method	Saves the game
Load	Method	Loads the game
Quit	Method	Quits the game

Save

Description

Saves the current game into a file. The file will be stored into the game directory and has the name SaveXX.lsd (XX will be replaced with the slot in two digit format). The scopes of the Destiny.dll won't be saved with this method. If the game can't save the RPG_RT.exe will display an error message and crash. Something more specific can be found at known bugs.

Syntax

```
Game.Save(Slot)
```

Return value

None

Type

Method

Parameter: Slot

Description

The number of the save slot.

Data type

Dword

Range

0 to 99

Example

```
$
Game.Save(1)
```

At the end the game whould be saved (except the destiny scopes) into the file Save01.lsd.

Load

Description

Loads the current game from a file. The file is stored into the game directory and has the name SaveXX.lsd (XX will be replaced with the slot in two digit format). The scopes of the Destiny.dll won't be loaded with this method. If the game can't be loaded the RPG_RT.exe will show an error message and crash. Something more specific can be found at known bugs.

Information *It's strongly recommended that you use an actual version of the Destiny.dll, because this command doesn't work reliably on older versions of Destiny.dll!*

Syntax

```
Game.Load(Slot)
```

Return value

None

Type

Method

Parameter: Slot

Description

The number of the save slot.

Data type

Dword

Range

0 to 99

Example

```
$
Game.Load(1)
```

At the end the game whould be loaded (except the destiny scopes) from the file Save01.lsd.

Quit

Description

Quits the current game and returns to Windows.

Syntax

```
Game.Quit()
```

Return value

None

Type

Method

Example

```
$
Game.Quit()
```

At the end the game whould exit.

9.3 Convert object

Description

You can convert data types with the Convert object.

List of methods/properties

Name	Type	Short description
DecimalComma	Property	Specifies wether a comma or a point is used for decimal separation
Byte	Method	Converts to the data type byte
Word	Method	Converts to the data type word
Dword	Method	Converts to the data type dword
Double	Method	Converts to the data type double
Switch	Method	Converts to the data type switch
String	Method	Converts to the data type string
Angle	Method	Converts between different angle formats

DecimalComma

Description

If this switch is activated then a comma is used for decimal separation instead of a point. This property affects only conversion from double to string.

Syntax

```
Convert.DecimalComma
```

Data type

Switch

Type

Property

Example

```
$
Convert.DecimalComma = True
```

Byte

Description

Converts each data type into the data type byte.

Syntax

```
Convert.Byte(Number)
```

Return value

Byte

Type

Method

Parameter: Number

Description

The number which should be converted.

Data type

All

Range

0 to 255

Example

```
$
d[1] = Convert.Byte(v[1])
```

Word

Description

Converts each data type into the data type word.

Syntax

```
Convert.Word(Number)
```

Return value

Word

Type

Method

Parameter: Number

Description

The number which should be converted.

Data type

All

Range

-32768 to 32767

Example

```
$
d[1] = Convert.Word(v[1])
```

Dword

Description

Converts each data type into the data type dword.

Syntax

```
Convert.Dword(Number)
```

Return value

Dword

Type

Method

Parameter: Number

Description

The number which should be converted.

Data type

All

Range

-2147483648 to 2147483647

Example

```
$
d[1] = Convert.Dword("1234")
```

d[1] whould be at end: 1234

Double

Description

Converts each data type into the data type double.

Syntax

```
Convert.Double(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which should be converted.

Data type

All

Range

-1.7E+308 to +1.7E+308

Example

```
$
f[1] = Convert.Double(v[1])
```

Switch

Description

Converts each data type into the data type switch.

Syntax

```
Convert.Switch(Number)
```

Return value

Switch

Type

Method

Parameter: Number

Description

The number which should be converted.

Data type

All

Range

0 to 1

Example

```
$
s[1] = Convert.Switch(v[1])
```

String

Description

Converts each data type into the data type string.

Syntax

```
Convert.String(Number)
```

Return value

String

Type

Method

Parameter: Number

Description

The number which should be converted.

Data type

All

Range

All valid numbers

Example

```
$
a[1] = Convert.String(v[1])
```

Angle

Description

Converts an angle from one format into an other. To specifiy the angle formats used you can use the angle format constants. (For a more specific description of the angle formats see Sin method of the Math object)

Syntax

```
Convert.Angle(Angle, FormatFrom, FormatTo)
```

Return value

Double

Type

Method

Parameter: Angle

Description

The angle which should be converted.

Data type

Double

Range

All valid angles

Parameter: FormatFrom

Description

The angle format which is currently used for the angle.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Parameter: FormatTo

Description

The angle format which shall be converted to.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
v[1] = Convert.Angle(90, DEG, RPG)
```

v[1] whould be at end: 64

9.4 Logic object

Description

With the Logic object you can apply logical operations with switches and comparisons with numbers. Additionally you can return different values conditioned by a switch via the If method.

List of methods/properties

Name	Type	Short description
Not	Method	Reverses the value of a switch
And	Method	Applys a logical AND operation with two switches
Or	Method	Applys a logical OR operations with two switches
Xor	Method	Applys a logical XOR operation with two switches
Xnor	Method	Applys a logical XNOR operation with two switches
Nand	Method	Applys a logical NAND operation with two switches
Nor	Method	Applys a logical NOR operation with two switches
Imp	Method	Applys a logical implication operation with two switches
Inh	Method	Applys a logical inhibit operation with two switches
And3	Method	Applys a logical AND operation with three switches
Or3	Method	Applys a logical OR operation with three switches
Xor3	Method	Applys a logical XOR operation with three switches
Xnor3	Method	Applys a logical XNOR operation with three switches
Nand3	Method	Applys a logical NAND operation with three switches
Nor3	Method	Applys a logical NOR operation with three switches
Above	Method	Checks whether a value is greater than an other value
AboveEqual	Method	Checks whether a value is greater/equal than an other value
Below	Method	Checks whether a value is smaller than an other value
BelowEqual	Method	Checks whether a value is smaller/equal than an other value
Equal	Method	Checks whether a value is equals to an other value

Unequal	Method	Checks whether a value isn't equal to an other value
If	Method	Returns one of two values conditioned by a switch

Not

Description

Returns the reversed value of a switch.

Syntax

```
Logic.Not(Switch1)
```

Truth table

Switch1	Return value
0	1
1	0

Return value

Switch

Type

Method

Parameter: Switch1

Description

The switch which shall be reversed.

Data type

Switch

Example

```
$
s[1] = Logic.Not(True)
```

s[1] whould be at end: 0 (= False)

And

Description

Applys a logical AND operation with two switches.

Syntax

```
Logic.And(Switch1, Switch2)
```

Truth table

Switch1	Switch2	Return value
0	0	0
0	1	0
1	0	0
1	1	1

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.And(True, False)
```

s[1] whould be at end: 0 (= False)

Or

Description

Applys a logical OR operation with two switches.

Syntax

```
Logic.Or(Switch1, Switch2)
```

Truth table

Switch1	Switch2	Return value
0	0	0
0	1	1
1	0	1
1	1	1

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Or(False, True)
```

s[1] whould be at end: 1 (= True)

Xor

Description

Applys a logical XOR operation with two switches.

Syntax

```
Logic.Xor(Switch1, Switch2)
```

Truth table

Switch1	Switch2	Return value
0	0	0
0	1	1
1	0	1
1	1	0

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Xor(True, True)
```

s[1] whould be at end: 0 (= False)

Xnor

Description

Applys a logical XNOR operation (equivalence) with two switches.

Syntax

```
Logic.Xnor(Switch1, Switch2)
```

Truth table

Switch1	Switch2	Return value
0	0	1
0	1	0
1	0	0
1	1	1

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Xnor(False, False)
```

s[1] whould be at end: 1 (= True)

Nand

Description

Applys a logical NAND operation with two switches.

Syntax

```
Logic.Nand(Switch1, Switch2)
```

Truth table

Switch1	Switch2	Return value
0	0	1
0	1	1
1	0	1
1	1	0

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Nand(True, True)
```

s[1] whould be at end: 0 (= False)

Nor

Description

Applys a logical NOR operation with two switches.

Syntax

```
Logic.Nor(Switch1, Switch2)
```

Truth table

Switch1	Switch2	Return value
0	0	1
0	1	0
1	0	0
1	1	0

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Nor(True, False)
```

s[1] whould be at end: 0 (= False)

Imp

Description

Applys a logical implication operation with two switches.

Syntax

```
Logic.Imp(Switch1, Switch2)
```

Truth table

Switch1	Switch2	Return value
0	0	1
0	1	1
1	0	0
1	1	1

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Imp(False, True)
```

s[1] whould be at end: 1 (= True)

Inh

Description

Applys a logical inhibit operation with two switches.

Syntax

```
Logic.Inh(Switch1, Switch2)
```

Truth table

Switch1	Switch2	Return value
0	0	0
0	1	1
1	0	0
1	1	0

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Inh(False, False)
```

s[1] whould be at end: 0 (= False)

And3

Description

Applys a logical AND operation with three switches.

Syntax

```
Logic.And3(Switch1, Switch2, Switch3)
```

Truth table

Switch1	Switch2	Switch3	Return value
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Parameter: Switch3

Description

The third switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.And3(True, True, True)
```

s[1] whould be at end: 1 (= True)

Or3

Description

Applys a logical OR operation with three switches.

Syntax

```
Logic.Or3(Switch1, Switch2, Switch3)
```

Truth table

Switch1	Switch2	Switch3	Return value
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Parameter: Switch3

Description

The third switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Or3(False, False, True)
```

s[1] whould be at end: 0 (= True)

Xor3

Description

Applys a logical XOR operation with three switches.

Syntax

```
Logic.Xor3(Switch1, Switch2, Switch3)
```

Truth table

Switch1	Switch2	Switch3	Return value
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Parameter: Switch3

Description

The third switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Xor3(True, True, True)
```

s[1] whould be at end: 0 (= False)

Xnor3

Description

Applys a logical XNOR operation (equivalence) with three switches.

Syntax

```
Logic.Xnor3(Switch1, Switch2, Switch3)
```

Truth table

Switch1	Switch2	Switch3	Return value
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Parameter: Switch3

Description

The third switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Xnor3(False, False, False)
```

s[1] whould be at end: 1 (= True)

Nand3

Description

Applys a logical NAND operation with three switches.

Syntax

```
Logic.Nand3(Switch1, Switch2, Switch3)
```

Truth table

Switch1	Switch2	Switch3	Return value
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Parameter: Switch3

Description

The third switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Nand3(True, True, False)
```

s[1] whould be at end: 1 (= True)

Nor3

Description

Applys a logical NOR operation with three switches.

Syntax

```
Logic.Nor3(Switch1, Switch2, Switch3)
```

Truth table

Switch1	Switch2	Switch3	Return value
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Return value

Switch

Type

Method

Parameter: Switch1

Description

The first switch which shall be operated.

Data type

Switch

Parameter: Switch2

Description

The second switch which shall be operated.

Data type

Switch

Parameter: Switch3

Description

The third switch which shall be operated.

Data type

Switch

Example

```
$
s[1] = Logic.Nor3(False, True, False)
```

s[1] whould be at end: 0 (= False)

Above

Description

Compares two numeric values an returns true if the first value is greater than the second value.

Syntax

```
Logic.Above(Number1, Number2)
```

Return value

Switch

Type

Method

Parameter: Number1

Description

The first number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Parameter: Number2

Description

The second number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Example

```
$
s[1] = Logic.Above(1, 7)
```

s[1] whould be at end: 0 (= False)

AboveEqual

Description

Compares two numeric values an returns true if the first value is greater/equal than the second value.

Syntax

```
Logic.AboveEqual(Number1, Number2)
```

Return value

Switch

Type

Method

Parameter: Number1

Description

The first number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Parameter: Number2

Description

The second number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Example

```
$
s[1] = Logic.AboveEqual(6, 6)
```

s[1] whould be at end: 1 (= True)

Below

Description

Compares two numeric values an returns true if the first value is smaller than the second value.

Syntax

```
Logic.Below(Number1, Number2)
```

Return value

Switch

Type

Method

Parameter: Number1

Description

The first number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Parameter: Number2

Description

The second number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Example

```
$
s[1] = Logic.Below(1, 20)
```

s[1] whould be at end: 1 (= True)

BelowEqual

Description

Compares two numeric values an returns true if the first value is smaller/equal than the second value.

Syntax

```
Logic.BelowEqual(Number1, Number2)
```

Return value

Switch

Type

Method

Parameter: Number1

Description

The first number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Parameter: Number2

Description

The second number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Example

```
$
s[1] = Logic.BelowEqual(71, 2)
```

s[1] whould be at end: 0 (= False)

Equal

Description

Compares two numeric values an returns true if the first value equals to the second value.

> ***Information***
> *This method compares only numbers. If you want to compare string then you must use the Compare method of the String object. If you want to compare switches then you must use the Xnor method of this object.*

Syntax

```
Logic.Equal(Number1, Number2)
```

Return value

Switch

Type

Method

Parameter: Number1

Description

The first number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Parameter: Number2

Description

The second number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Example

```
$
s[1] = Logic.Equal(7, 6)
```

s[1] whould be at end: 0 (= False)

Unequal

Description

Compares two numeric values an returns true if the first value doesn't equal to the second value.

> ***Information***
> *This method compare only numbers. If you want to compare strings for inequality then you must use the Compare method of the String object and reverse the result with the Not method of this object. If you want to compare switches for inequality then you must use the Xor method of this object.*

Syntax

```
Logic.Unequal(Number1, Number2)
```

Return value

Switch

Type

Method

Parameter: Number1

Description

The first number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Parameter: Number2

Description

The second number which shall be compared.

Data type

All numbers

Range

Depends on the data type

Example

```
$
s[1] = Logic.Unequal(12, 8)
```

s[1] whould be at end: 1 (= True)

If

Description

Returns a value conditioned by the value of a switch. If the switch is True then TrueValue will be returned, otherwise FalseValue will be returned.

> ***Information***
> *The values including the data types of the parameters will be returned. However you can't use this method for a writing operation.*

Syntax

```
Logic.If(Expression, TrueValue, FalseValue)
```

Return value

Either TrueValue or FalseValue (depends on the value of Expression).

Type

Method

Parameter: Expression

Description

Decides whether TrueValue or FalseValue returns.

Data type

Switch

Parameter: TrueValue

Description

Will be returned only if Expression is true.

Data type

All

Parameter: FalseValue

Description

Will be returned only if Expression is false.

Data type

All

Example

```
$
a[1] = Logic.If(True, "marmalade", "cake")
```

a[1] whould be at end: "marmalade"

9.5 Math object

Description

You can apply many mathematical operations (trigonometry, logarithmize, square root, rounding, ...) with the Math object.

List of methods/properties

Name	Type	Short description
Pi	Property	Returns Ludolph's number (π = 3,141592653589...)
E	Property	Returns Euler's number (e = 2,718281828...)
Abs	Method	Returns the absolute value of a number
Sin	Method	Calculates the sine of an angle
Cos	Method	Calculates the cosine of an angle
Tan	Method	Calculates the tangent of an angle
Cot	Method	Calculates the cotangent of an angle
Sec	Method	Calculates the sekant of an angle
Csc	Method	Calculates the cosekant of an angle
ASin	Method	Calculates the angle of a sine
ACos	Method	Calculates the angle of a cosine
ATan	Method	Calculates the angle of a tangent
ACot	Method	Calculates the angle of a cotangent
ASec	Method	Calculates the angle of a sekant
ACsc	Method	Calculates the angle of a cosekant
SinH	Method	Calculates the hyperbolic sine
CosH	Method	Calculates the hyperbolic cosine
TanH	Method	Calculates the hyperbolic tangent
CotH	Method	Calculates the hyperbolic cotangent
SecH	Method	Calculates the hyperbolic sekant
CscH	Method	Calculates the hyperbolic cosekant
ASinH	Method	Calculates the inverted hyperbolic sine
ACosH	Method	Calculates the inverted hyperbolic cosine
ATanH	Method	Calculates the inverted hyperbolic tangent
ACotH	Method	Calculates the inverted hyperbolic cotangent
ASecH	Method	Calculates the inverted hyperbolic sekant

ACscH	Method	Calculates the inverted hyperbolic cosekant
Power	Method	Exponentiates a base with an exponent
Log	Method	Calculates the logarithm to any base
Lg	Method	Calculates the decade logarithm (base: 10)
Ln	Method	Calculates the natural logarithm (base: e = 2,718281828...)
Lb	Method	Calculates the binary logarithm (base: 2)
Sqrt	Method	Calculates the square root of a number
Cmp	Method	Compares two values
Exp	Method	Exponates the number 10 with an exponent
Round	Method	Rounds a number halfway away from zero
RoundUp	Method	Rounds a number in direction to +∞
RoundDown	Method	Rounds a number in direction to -∞
Int	Method	Cuts the decimal places of a number
Scale	Method	Cuts the integer places of a number

Pi

Description

This property represents Ludolph's number (π = 3,141592653589...).

Syntax

```
Math.Pi
```

Data type

Double

Type

Property, read-only

Example

```
$
f[1] = Math.Pi
```

f[1] whould be at end: 3,141592653589...

E

Description

This property represents Euler's number (e = 2,718281828...).

Syntax

```
Math.E
```

Data type

Double

Type

Property, read-only

Example

```
$
f[1] = Math.E
```

f[1] whould be at end: 2,718281828...

Abs

Description

Returns the absolute value of a number (this means the leading sign will always be plus).

Syntax

```
Math.Abs(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number whose absolute value shall be returned.

Data type

Double

Example

```
$
f[1] = Math.Abs(-2)
```

f[1] whould be at end: 2

Sin

Description

Calculates the sine of an angle (= opposite leg / hypotenuse). You can specify the angle in one of four angle formats: DEG, RAD, GRAD and RPG. These names are specified as constants and can be used directly as parameter. DEG stands for degree and means that a full circle has 360 angle units. RAD stands for radiant and means that the radian measure (a full circle has π angle units) is used. GRAD stands for grad and means that a full circle has 400 angle units. RPG is a RPG-Maker specific format and means that a full circle has 256 angle units.

Syntax

```
Math.Sin(Angle, Format)
```

Return value

Double

Type

Method

Parameter: Angle

Description

The angle which shall be calculated.

Data type

Double

Parameter: Format

Description

The current format of the angle.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Sin(90, DEG)
```

f[1] whould be at end: 1

Cos

Description

Calculates the cosine of an angle (= adjacent leg / hypotenuse). (For a description of the angle formats see Sin method)

Syntax

```
Math.Cos(Angle, Format)
```

Return value

Double

Type

Method

Parameter: Angle

Description

The angle which shall be calculated.

Data type

Double

Parameter: Format

Description

The current format of the angle.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Cos(Math.Pi, RAD)
```

f[1] whould be at end: -1

Tan

Description

Calculates the tangent of an angle (= opposite leg / adjacent leg). (For a description of the angle formats see Sin method)

Syntax

```
Math.Tan(Angle, Format)
```

Return value

Double

Type

Method

Parameter: Angle

Description

The angle which shall be calculated.

Data type

Double

Parameter: Format

Description

The current format of the angle.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Tan(50, GRAD)
```

f[1] whould be at end: 1

Cot

Description

Calculates the cotangent of an angle (= adjacent leg / opposite leg). (For a description of the angle formats see Sin method)

Syntax

```
Math.Cot(Angle, Format)
```

Return value

Double

Type

Method

Parameter: Angle

Description

The angle which shall be calculated.

Data type

Double

Parameter: Format

Description

The current format of the angle.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Cot(96, RPG)
```

f[1] whould be at end: -1

Sec

Description

Calculates the sekant of an angle (= hypotenuse / adjacent leg). (For a description of the angle formats see Sin method)

Syntax

```
Math.Sec(Angle, Format)
```

Return value

Double

Type

Method

Parameter: Angle

Description

The angle which shall be calculated.

Data type

Double

Parameter: Format

Description

The current format of the angle.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Sec(45, DEG)
```

f[1] whould be at end: 1,414213562373...

Csc

Description

Calculates the cosekant of an angle (= hypotenuse / opposite leg). (For a description of the angle formats see Sin method)

Syntax

```
Math.Csc(Angle, Format)
```

Return value

Double

Type

Method

Parameter: Angle

Description

The angle which shall be calculated.

Data type

Double

Parameter: Format

Description

The current format of the angle.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Csc(30, DEG)
```

f[1] whould be at end: 2

ASin

Description

Calculates the angle from a sine (= arc sine). (For a description of the angle formats see Sin method)

Syntax

```
Math.Asin(Sine, Format)
```

Return value

Double

Type

Method

Parameter: Sine

Description

The sine value.

Data type

Double

Parameter: Format

Description

The target angle format.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Asin(30, DEG)
```

f[1] whould be at end: 30

ACos

Description

Calculates the angle from a cosine (arc cosine). (For a description of the angle formats see Sin method)

Syntax

```
Math.Acos(Cosine, Format)
```

Return value

Double

Type

Method

Parameter: Cosine

Description

The cosine value.

Data type

Double

Parameter: Format

Description

The target angle format.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Acos(1, GRAD)
```

f[1] whould be at end: 0

ATan

Description

Calculates the angle from a tangent (= arc tangent). (For a description of the angle formats see Sin method)

Syntax

```
Math.Atan(Tangent, Format)
```

Return value

Double

Type

Method

Parameter: Tangent

Description

The tangent value.

Data type

Double

Parameter: Format

Description

The target angle format.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Atan(1, DEG)
```

f[1] whould be at end: 45

ACot

Description

Calculates the angle from a cotangent (= arc cotangent). (For a description of the angle formats see Sin method)

Syntax

```
Math.Acot(Cotangent, Format)
```

Return value

Double

Type

Method

Parameter: Cotangent

Description

The cotangent value.

Data type

Double

Parameter: Format

Description

The target angle format.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Acot(-1, DEG)
```

f[1] whould be at end: 135 (= -45)

ASec

Description

Calculates the angle from a sekant (= arc sekant). (For a description of the angle formats see Sin method)

Syntax

```
Math.Asec(Sekant, Format)
```

Return value

Double

Type

Method

Parameter: Sekant

Description

The sekant value.

Data type

Double

Parameter: Format

Description

The target angle format.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Asec(2, DEG)
```

f[1] whould be at end: 60

ACsc

Description

Calculates the angle from a cosekant (= arc cosekant). (For a description of the angle formats see Sin method)

Syntax

```
Math.Acsc(Cosekant, Format)
```

Return value

Double

Type

Method

Parameter: Cosekant

Description

The cosekant value.

Data type

Double

Parameter: Format

Description

The target angle format.

Data type

Dword

Range

1 to 4 (Constants: DEG, RAD, GRAD and RPG)

Example

```
$
f[1] = Math.Acsc(1, RPG)
```

f[1] whould be at end: 64

SinH

Description

Calculates the hyperbolic sine.

Syntax

```
Math.Sinh(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Sinh(f[2])
```

CosH

Description

Calculates the hyperbolic cosine.

Syntax

```
Math.Cosh(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Cosh(f[2])
```

TanH

Description

Calculates the hyperbolic tangent.

Syntax

```
Math.Tanh(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Tanh(f[2])
```

CotH

Description

Calculates the hyperbolic cotangent.

Syntax

```
Math.Coth(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Coth(f[2])
```

SecH

Description

Calculates the hyperbolic sekant.

Syntax

```
Math.Sech(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Sech(f[2])
```

CscH

Description

Calculates the hyperbolic cotangent.

Syntax

```
Math.Csch(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Csch(f[2])
```

ASinH

Description

Calculates the inverted hyperbolic sine (area hyperbolic sine).

Syntax

```
Math.Asinh(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Asinh(f[2])
```

ACosH

Description

Calculates the inverted hyperbolic cosine (area hyperbolic cosine).

Syntax

```
Math.Acosh(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Acosh(f[2])
```

ATanH

Description

Calculates the inverted hyperbolic tangent (area hyperbolic tangent).

Syntax

```
Math.Atanh(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Atanh(f[2])
```

ACotH

Description

Calculates the inverted hyperbolic cotangent (area hyperbolic cotangent).

Syntax

```
Math.Acoth(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Acoth(f[2])
```

ASecH

Description

Calculates the inverted hyperbolic sekant (area hyperbolic sekant).

Syntax

```
Math.Asech(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Asech(f[2])
```

ACscH

Description

Calculates the inverted hyperbolic cosekant (area hyperbolic cosekant).

Syntax

```
Math.Acsch(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Acsch(f[2])
```

Power

Description

Exponentiates the base with the exponent.

Syntax

```
Math.Power(Base, Exponent)
```

Return value

Double

Type

Method

Parameter: Base

Description

The base value.

Data type

Double

Parameter: Exponent

Description

The exponent value.

Data type

Double

Example

```
$
f[1] = Math.Power(3, 4)
```

f[1] whould be at end: 81 (= 3 * 3 * 3 * 3)

Log

Description

Calculates the logarithm of any base.

Syntax

```
Math.Log(Number, Base)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number whose logarithm shall be calculated.

Data type

Double

Parameter: Base

Description

The base of the logarithm.

Data type

Double

Example

```
$
f[1] = Math.Log(25, 5)
```

f[1] whould be at end: 2

Lg

Description

Calculates the decade logarithm (the base is 10).

Syntax

```
Math.Lg(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number whose logarithm shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Lg(1000)
```

f[1] whould be at end: 3

Ln

Description

Calculates the natural logarith (the base is Euler's number).

Syntax

```
Math.Ln(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number whose logarithm shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Ln(1 / Math.E)
```

f[1] whould be at end: -1

Lb

Description

Calculates the binary logarithm (the base is 2).

Syntax

```
Math.Lb(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number whose logarithm shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Lb(256)
```

f[1] whould be at end: 8

Sqrt

Description

Calculates the square root of a number.

Syntax

```
Math.Sqrt(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number whose square root shall be calculated.

Data type

Double

Example

```
$
f[1] = Math.Sqrt(10000)
```

f[1] whould be at end: 100

Cmp

Description

Compares two numbers. If the first number is smaller than the second number then the result will be < 0. If the first number is greater than the second number then the result will be > 0. If both numbers are the same then the result will be = 0.

Syntax

```
Math.Cmp(Number1, Number2)
```

Return value

Dword

Type

Method

Parameter: Number1

Description

The first number to compare.

Data type

Double

Parameter: Number2

Description

The second number to compare.

Data type

Double

Example

```
$
d[1] = Math.Cmp(1, 2)
```

f[1] whould be at end: -1

Exp

Description

Exponentiates the number 10 with the specified number.

Syntax

```
Math.Exp(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The exponent for the number 10.

Data type

Double

Example

```
$
f[1] = Math.Exp(5)
```

f[1] whould be at end: 100000

Round

Description

Rounds a number halfway away from zero (this means 5 to 9 will be rounded up) to the specified place. If you specify 0 as place then the decimal part will be rounded to the integer part. If you specify a positive number as place then you round to that decimal place. If you specify a negative number as place then you round to that integer place.

Syntax

```
Math.Round(Number, Place)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be rounded.

Data type

Double

Parameter: Place

Description

The place which shall be rounded (measured from the decimal separator).

Data type

Dword

Example

```
$
f[1] = Math.Round(3.345, 2)
```

f[1] whould be at end: 3.35

RoundUp

Description

Rounds a number into the direction of +∞ (this means from 1 to 9 will be rounded up) to the specified place. If you specify 0 as place then the decimal part will be rounded to the integer part. If you specify a positive number as place then you round to that decimal place. If you specify a negative number as place then you round to that integer place.

Syntax

```
Math.RoundUp(Number, Place)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be rounded.

Data type

Double

Parameter: Place

Description

The place which shall be rounded (measured from the decimal separator).

Data type

Dword

Example

```
$
f[1] = Math.RoundUp(-300.7, 0)
```

f[1] whould be at end: -300 (if you round into the direction of +∞ then the result will always be more positive)

RoundDown

Description

Rounds a number into the direction of -∞ (this means from 1 to 9 will be rounded down) to the specified place. If you specify 0 as place then the decimal part will be rounded to the integer part. If you specify a positive number as place then you round to that decimal place. If you specify a negative number as place then you round to that integer place.

Syntax

```
Math.RoundDown(Number, Place)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number which shall be rounded.

Data type

Double

Parameter: Place

Description

The place which shall be rounded (measured from the decimal separator).

Data type

Dword

Example

```
$
f[1] = Math.RoundDown(592.001, -2)
```

f[1] whould be at end: 500

Int

Description

Cuts the decimal part of a number.

Syntax

```
Math.Int(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number whose decimal part shall be removed.

Data type

Double

Example

```
$
f[1] = Math.Int(123.456)
```

f[1] whould be at end: 123

Scale

Description

Cuts the integer part of a number.

Syntax

```
Math.Scale(Number)
```

Return value

Double

Type

Method

Parameter: Number

Description

The number whose integer part shall be removed.

Data type

Double

Example

```
$
f[1] = Math.Scale(123.456)
```

f[1] whould be at end: 0,456

9.6 String object

Description

You can manipulate (cut, search, compare, ...) strings with the String object.

Liste of methods/properties

Name	Type	Short description
Length	Method	Returns the length of a string
LTrim	Method	Truncates spaces from the left side of a string
RTrim	Method	Truncates spaces from the right side of a string
Trim	Method	Truncates spaces from both sides (left and right) of a string
Chr	Method	Translates an ASCII code to a char
Ord	Method	Returns the ASCII code of a char
Pos	Method	Returns the position of a search string
SubStr	Method	Returns a specified part of a string
Compare	Method	Compares two strings
Replace	Method	Replaces all occurences of a string in an other string
ToUpper	Method	Translates each letter into upper case letters
ToLower	Method	Translates each letter into lower case letters
Reverse	Method	Reverses each char of a string
Fill	Method	Concatenates a string multiple times
Format	Method	Formats numbers
WeekdayName	Method	Returns the name of a weekday
MonthName	Method	Returns the name of a month

Length

Description

Returns the number of chars in a string.

Syntax

```
String.Length(String)
```

Return value

Dword

Type

Method

Parameter: String

Description

The string whose length shall be determined.

Data type

String

Example

```
$
d[1] = String.Length("7 Chars")
```

d[1] whould be at end: 7

LTrim

Description

Truncates spaces (this means chars with the ASCII code 32) from the left side of a string.

Syntax

```
String.LTrim(String)
```

Return value

String

Type

Method

Parameter: String

Description

The string whose spaces shall be removed from the left side.

Data type

String

Example

```
$
a[1] = String.Trim(" Text with spaces left and right ")
```

a[1] whould be at end: "Text with spaces left and right "

RTrim

Description

Truncates spaces (this means chars with the ASCII code 32) from the right side of a string.

Syntax

```
String.RTrim(String)
```

Return value

String

Type

Method

Parameter: String

Description

The string whose spaces shall be removed from the right side.

Data type

String

Example

```
$
a[1] = String.Trim(" Text with spaces left and right ")
```

a[1] whould be at end: " Text with spaces left and right"

Trim

Description

Truncates spaces (this means chars with the ASCII code 32) from both sides (left and right) of a string.

Syntax

```
String.Trim(String)
```

Return value

String

Type

Method

Parameter: String

Description

The string whose spaces shall be removed from the left and right side.

Data type

String

Example

```
$
a[1] = String.Trim(" Text with spaces left and right ")
```

a[1] whould be at end: "Text with spaces left and right"

Chr

Description

Creates a string from an ASCII code.

Syntax

```
String.Chr(Char)
```

Return value

String

Type

Method

Parameter: Char

Description

The ASCII code of the char. The ASCII code 0 isn't valid.

Data type

Byte

Example

```
$
a[1] = String.Chr(65)
```

a[1] whould be at end: "A"

Ord

Description

Returns the ASCII code of a char at a specified position in a string.

Syntax

```
String.Ord(String, Position)
```

Return value

Byte

Type

Method

Parameter: String

Description

The string which contains the char.

Data type

String

Parameter: Position

Description

The position of the char in the string. This is the offset from the start of the string (this means 0 whould be the first char, 1 whould be the second char, ...).

Data type

Dword

Example

```
$
d[1] = String.Ord("Text", 2)
```

d[1] whould be at end: 120 (this is the ASCII code of the char "x")

Pos

Description

Returns the position of a partial string in an other string. The return value is the offset from the start of the string (this means 0 whould be the first char, 1 whould be the second char, ...). This method returns -1 if the string couldn't be found.

Syntax

```
String.Pos(String, SearchString, StartPosition)
```

Return value

Dword

Type

Method

Parameter: String

Description

The string which contains SearchString (haystack).

Data type

String

Parameter: SearchString

Description

The string which shall be searched (needle).

Data type

String

Parameter: StartPosition

Description

The position where the search shall start. This value is the offset from the start of the string (this means 0 whould start at the first char, 1 whould start at the second char, ...).

Data type

Dword

Example

```
$
d[1] = String.Pos("Search me!", "e", 2)
```

d[1] whould be at end: 7 (the first "e" has been skipped because the search started at char no. 3)

SubStr

Description

Returns a partial string with specified length at a specified position.

Syntax

```
String.SubStr(String, Position, Length)
```

Return value

String

Type

Method

Parameter: String

Description

The string which contains the partial string.

Data type

String

Parameter: Position

Description

The start position where of the partial string. This value is the offset from the start of the string (this means 0 whould start at the first char, 1 whould start at the second char, ...).

Data type

Dword

Parameter: Length

Description

The length of the partial string. If this value is greater than the remainder of the string (or if it is a negative value) then the entire remainder will be copied.

Data type

Dword

Example

```
$
a[1] = String.SubStr("Text with parts", 5, 4)
```

a[1] whould be at end: "with"

Compare

Description

Compares two strings and returns true if they are equal (otherwise false).

Syntax

```
String.Compare(String1, String2)
```

Return value

Switch

Type

Method

Parameter: String1

Description

The first string which shall be compared.

Data type

String

Parameter: String2

Description

The second string which shall be compared.

Data type

String

Example

```
$
s[1] = String.Compare("Text", "Text")
```

s[1] whould be at end: 1 (= True)

Replace

Description

Replaces all occurences of a partial string in an other string.

Syntax

```
String.Compare(Expression, Search,
Replacement)
```

Return value

String

Type

Method

Parameter: Expression

Description

The string which contains Search (Haystack).

Data type

String

Parameter: Search

Description

The string which shall be replaced (Needle).

Data type

String

Parameter: Replacement

Description

The string which shall be used as replacement for Search.

Data type

String

Example

```
$
a[1] = String.Replace("Milk products are
from cows",
"Milk", "Meat")
```

a[1] whould be at end: "Meat products are from cows"

ToUpper

Description

Translates each letter (a to z) of a String into upper case letters.

Syntax

```
String.ToUpper(String)
```

Return value

String

Type

Method

Parameter: String

Description

The string whose lower case letters shall be translated into upper case letters.

Data type

String

Example

```
$
a[1] = String.ToUpper("UPPER and lower case LETTERS")
```

a[1] whould be at end: "UPPER AND LOWER CASE LETTERS"

ToLower

Description

Translates each letter (a to z) of a string into lower case letters.

Syntax

```
String.ToLower(String)
```

Return value

String

Type

Method

Parameter: String

Description

The string whose upper case letters shall be translated into lower case letters.

Data type

String

Example

```
$
a[1] = String.ToLower("UPPER and lower case LETTERS")
```

a[1] whould be at end: "upper and lower case letters"

Reverse

Description

Reverses the content of a string.

Syntax

```
String.Reverse(String)
```

Return value

String

Type

Method

Parameter: String

Description

The string whose content shall be reversed.

Data type

String

Example

```
$
a[1] = String.Reverse("Reversed")
```

a[1] whould be at end: "desreveR"

Fill

Description

Concatenates a string multiple times.

Syntax

```
String.Fill(String, Count)
```

Return value

String

Type

Method

Parameter: String

Description

The string which shall be repeated.

Data type

String

Parameter: Count

Description

The number of repeats.

Data type

Dword

Range

0 to 10000

Example

```
$
a[1] = "We are" + String.Fill(" hungry", 3)
```

a[1] whould be at end: "We are hungry hungry hungry"

Format

Description

Formats a number similar to the MessageLink. The first char must be a F (for double) or a D (for dword). Accordingly the second parameter will be formatted either as dword or double. If it is formatted as dword then the minimum length of digits can follow the D (e. g. D4 whould be a dword with at least 4 digits). If the number is formatted as double then you can specify the minimum integer length and/or the exact decimal length (e. g. F2.3 for at least 2 integer digits an exact 3 decimal digits).

Syntax

```
String.Format(Format, Number)
```

Return value

String

Type

Method

Parameter: Format

Description

The format string for the number.

Data type

String

Parameter: Number

Description

The number which shall be formatted.

Data type

Dword or double

Example

```
$
a[1] = String.Format("f0.4", 123.7)
```

a[1] whould be at end: "123.7000" (if Convert.DecimalComma whould be 1 then a comma whould have been used instead of a point as decimal separator)

WeekdayName

Description

Returns the name of a weekday. This method depends on the chosen language of Destiny.dll.

Syntax

```
String.WeekdayName(Number, Short)
```

Return value

String

Type

Method, depends on language

Parameter: Number

Description

The number of the weekday. (0 = Sunday, 1 = Monday, 2 = Tuesday, 3 = Wednesday, 4 = Thursday, 5 = Friday, 6 = Saturday)

Data type

Byte

Range

0 to 6

Parameter: Short

Description

Specifies either the weekday shall return in short format (e. g. Sat) or long format (e. g. Saturday). True means short format.

Data type

Switch

Example

```
$
a[1] = String.WeekdayName(3, True)
```

a[1] whould be at end: "Wed" (if Destiny.Language whould be 0 then the return value whould be "Mi")

MonthName

Description

Returns the name of a month. This method depends on the chosen language of Destiny.dll.

Syntax

```
String.MonthName(Number, Short)
```

Return value

String

Type

Method, depends on language

Parameter: Number

Description

The number of the month. (1 = January, 2 = February, 3 = March, 4 = April, 5 = May, 6 = June, 7 = July, 8 = August, 9 = September, 10 = October, 11 = November, 12 = December)

Data type

Byte

Range

1 to 12

Parameter: Short

Description

Specifies either the month shall return in short format (e. g. Jan) or long format (e. g. January). True means short format.

Data type

Switch

Example

```
$
a[1] = String.MonthName(12, False)
```

a[1] whould be at end: "December" (if Destiny.Language whould be 1 then the return value whould be "Dezember")

9.7 Error object

Description

You can enable/disable single error messages and/or query captions with the Error object. This object requires an index to specify which error message shall be responded. You can use the Error constants for this index (e. g. Error[ERROR_SYNTAX].Enabled). A description of the errors can be found at the error messages.

List of methods/properties

Name	Type	Short description
Enabled	Property	Specifies wether an error message is enabled
Title	Property	The title of an error message
Message	Property	The content of an error message

Enabled

Description

If this switch is activated then the error message will be displayed if necessary. If all error messages have been disabled using the Errors object this switch is ineffective.

Syntax

```
Error[Index].Enabled
```

Data type

Switch

Type

Property

Example

```
$
Error[ERROR_READONLY].Enabled = False
```

At the end the error message for the write access on read-only values whould be disabled.

Title

Description

This property returns the title of an error message used in the title of the error message window. This value depends on the chosen language of the Destiny.dll.

Syntax

```
Error[Index].Title
```

Data type

String

Type

Property, read-only, depends on language

Example

```
$
a[1] = Error[ERROR_SYNTAX].Title
```

a[1] whould be at the end: "Error 1: Syntax" (if Destiny.Language whould be 0 then a[1] whould be a german title)

Message

Description

This property returns the message used in the body of the error message window. This value depends on the chosen language of the Destiny.dll.

Syntax

```
Error[Index].Message
```

Data type

String

Type

Property, read-only, depends on language

Example

```
$
a[1] = Error[ERROR_SYNTAX].Message
```

a[1] whould be at the end: "The syntax is invalid!" (if Destiny.Language whould be 0 then a[1] whould be a german message)

9.8 Errors object

Description

You can enable/disable all error messages with the Error object.
Additionally you control the error handling.

List of methods/properties

Name	Type	Short description
Enable	Method	Enables the error message output system
Disable	Method	Disables the error message output system
Resume	Method	Admits multi-line DestinyScripts to continue on errors
Halt	Method	Admits multi-line DestinyScript to abort on errors
Catch	Method	Returns the number of the last occured error

Enable

Description

Enables the error message output system. This has no effect on the single disabled errors via the Error object (this means if an error occurs and it has been disabled with the Error object then there won't be any error message shown).

Syntax

```
Errors.Enable()
```

Return value

None

Type

Method

Example

```
$
Errors.Enable()
```

At the end error messages whould be shown if necessary (= default option).

Disable

Description

Disables the error message output system. All error messages will be suppressed. This happens even if single errors have been enabled using the Error object.

Syntax

```
Errors.Disable()
```

Return value

None

Type

Method

Example

```
$
Errors.Disable()
```

At the end no error messages whould be shown.

Resume

Description

If you call this method then multi-line DestinyScripts will continue running if an error occurs.

Syntax

```
Errors.Resume()
```

Return value

None

Type

Method

Example

```
1 $
2 v[1] = 0;
3 Errors.Resume();
4 v[2] = 5 / 0;
5 v[1] = 5
```

v[1] whould be at end: 5 (the line 4 raises a "Division by zero" error. The following lines will still be executed!)

Halt

Description

If you call this method then multi-line DestinyScripts will abort if an error occurs (= default option).

Syntax

```
Errors.Halt()
```

Return value

None

Type

Method

Example

```
1 $
2 v[1] = 0;
3 Errors.Halt();
4 v[2] = 5 / 0;
5 v[1] = 5
```

v[1] whould be at end: 0 (the line 4 raises a "Division by zero" error. So the following lines won't be executed!)

Catch

Description

This method returns the number of the last occured error. If no error has occured the return value is 0. After a query of this method the number of the last occured error will be reset to 0. If an unknown error occured the return value is -1 (this differs from ERROR_UNKNOWN which has the value 0).

Syntax

```
Errors.Catch()
```

Return value

Dword

Type

Method

Example

In this example two DestinyScripts will run serially, but in two different RPG-Maker comments.

```
$
v[2] /= 0
```

```
$
v[1] = Errors.Catch()
```

v[1] whould be at end: 15 (this is the error number of the "Division by zero" number)

9.9 Keyboard object

Description

You can query/set key states with the Keyboard object. The mouse buttons (left, middle, right) will be queried with this object, too. (You can use the mouse button constants for this)

List of methods/properties

Name	Type	Short description
GetKeyState	Method	Queries the state of a key
GetKey	Method	Returns the key code of the last pressed key
GetKeyText	Method	Returns the key code of the last pressed key considering to the char repeat
SetKeyState	Method	Sets the state of a key

GetKeyState

Description

With this method you can query the current key state of a specified key. You can use the virtual key code constants for this. This method returns a value unequal to zero if the key is pressed. This method is equivalent to the Windows function GetAsyncKeyState.

Syntax

```
Keyboard.GetKeyState(Keycode)
```

Return value

Word

Type

Method

Parameter: Keycode

Description

The number of the key to be queried. You can use the virutal key code constants for this.

Datentyp

Dword

Example

```
$
v[1] = Keyboard.GetKeyState(VK_DOWN)
```

If the key [Arrow down] is pressed then v[1] whould be at end -32767 or -32768, otherwise 0 or 1.

GetKey

Description

Queries all keys from 1 to 254 and returns the first number of the pressed key. If no key is pressed this method will return 0.

Syntax

```
Keyboard.GetKey()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = Keyboard.GetKey()
```

At the end the virtual key code of the last pressed key whould be returned. But only the first found key will be returned. If more than one key is pressed at the same time then only the lower virtual key code will be returned. Hence you should use the GetKeyText method for text input.

GetKeyText

Description

Queries all keys from 1 to 254 and returns the first number of the pressed key considering to the char repeat. If no key is pressed with expedient char repeat this method will return 0. This method can be used for text input.

Syntax

```
Keyboard.GetKeyText()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = Keyboard.GetKeyText()
```

At the end the last pressed key (considering to the char repeat) whould be returned. If you use this in a loop then you could input chars in the correct order. If a char whould be hold down then the char repeat whould make sure that not each loop will return this key code.

SetKeyState

Description

You can set the state of a key with this method. You can use the virtual key code constants for this. To specify the key state you can use the key state constants. This method is equivalent to the windows function keybd_event.

Syntax

```
Keyboard.SetKeyState(Keycode, Keystate)
```

Return value

None

Type

Method

Parameter: Keycode

Description

The virtual key code. You can use the virtual key code constants for this.

Datentyp

Dword

Parameter: Keystate

Description

The new key state. You can use the key state constants for this.

Datentyp

Dword

Example

```
$
Keyboard.SetKeyState(VK_RIGHT, KEYEVENTF_KEYDOWN)
```

At the end the player whould try to move right, because the computer thinks the right arrow key is pressed. This will stop if the right arrow key is released (in this case you must even press it first!). To "release" the key via DestinyScript you can use the KEYEVENTF_KEYUP constant instead of KEYEVENTF_KEYDOWN.

9.10 Mouse object

Description

You can get/set the position of the mouse cursor via the Mouse object. The cursor position is relative to the upper left corner of the game window. If the game is in window mode the coordinates will be transformed automatically.

List of methods/properties

Name	Type	Short description
X	Property	The current x coordinate of the mouse cursor
Y	Property	The current y coordinate of the mouse cursor

X

Description

This property represents the x coordinate of the mouse cursor and is relative to the upper left corner of the game window. If the game window is stretched then the x coordinate will be transformed automatically. The unit for this value is pixel.

Syntax

```
Mouse.X
```

Data type

Dword

Type

Property

Example

```
$
Mouse.X = 10
```

At the end the x coordinate of the mouse cursor whould be 10 pixel away from the left border of the game window.

Y

Description

This property represents the y coordinate of the mouse cursor and is relative to the upper left corner of the game window. If the game window is stretched then the y coordinate will be transformet automatically. The unit for this value is pixel.

Syntax

```
Mouse.Y
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Mouse.Y
```

At the end v[1] whould be the current y coordinate relative to the upper border of the game window.

9.11 Time object

Description

You can query the current date/time of the computer with the Time object.

List of methods/properties

Name	Type	Short description
Weekday	Property	The current weekday
Day	Property	The current day of the month
Month	Property	The current month
Year	Property	The current year
Hour	Property	The current hour
Minute	Property	The current minute
Second	Property	The current second
Millisecond	Property	The current millisecond
Tick	Property	A continuous counter in milliseconds

Weekday

Description

This property returns the current weekday. Sunday is the first day of the week (0 = Sunday, 1 = Monday, 2 = Tuesday, 3 = Wednesday, 4 = Thursday, 5 = Friday, 6 = Saturday).

Syntax

```
Time.Weekday
```

Data type

Word

Type

Property, read-only

Example

```
$
v[1] = Time.Weekday
```

At the end v[1] whould be the current weekday (e. g. on a tuesday v[1] whould be 2).

Day

Description

This property returns the current day of the month.

Syntax

```
Time.Day
```

Data type

Word

Type

Property, read-only

Example

```
$
v[1] = Time.Day
```

At the end v[1] whould be the current day of the month (e. g. at the 08. January 1987 this whould be 8).

Month

Description

This property returns the current month. (1 = January, 2 = February, 3 = March, 4 = April, 5 = May, 6 = June, 7 = July, 8 = August, 9 = September, 10 = October, 11 = November, 12 = December)

Syntax

```
Time.Month
```

Data type

Word

Type

Property, read-only

Example

```
$
v[1] = Time.Month
```

At the end v[1] whould be the current month (e. g. at the 08. January 1987 this whould be 1).

Year

Description

This property returns the current year.

Syntax

```
Time.Year
```

Data type

Word

Type

Property, read-only

Example

```
$
v[1] = Time.Year
```

At the end v[1] whould be the current year (e. g. at the 08. January 1987 this whould be 1987).

Hour

Description

This property returns the current hour.

Syntax

```
Time.Hour
```

Data type

Word

Type

Property, read-only

Example

```
$
v[1] = Time.Hour
```

At the end v[1] whould be the current hour (e. g. at 12:30 this whould be 12).

Minute

Description

This property returns the current minute.

Syntax

```
Time.Minute
```

Data type

Word

Type

Property, read-only

Example

```
$
v[1] = Time.Minute
```

At the end v[1] whould be the current minute (e. g. at 12:30 this whould be 30).

Second

Description

This property returns the current second.

Syntax

```
Time.Second
```

Data type

Word

Type

Property, read-only

Example

```
$
v[1] = Time.Second
```

At the end v[1] whould be the current second.

Millisecond

Description

This property returns the current millisecond.

Syntax

```
Time.Millisecond
```

Data type

Word

Type

Property, read-only

Example

```
$
v[1] = Time.Millisecond
```

At the end v[1] whould be the current millisecond.

Tick

Description

This counter counts every millisecond up by one. The value of this property is equivalent to the Windows function GetTickCount. This property can be used to measure time differences. You simply save the value before and after an action. The difference is the required time for that action in milliseconds.

Syntax

```
Time.Tick
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Time.Tick
```

At the end v[1] whould be the current tick of the computer.

9.12 Actor object

Description

You can get/set the properties of an hero with the Actor object. This object requires an index to specify which hero shall be responded. This index is the same as the hero id in the RPG-Maker 2000 database.

List of methods/properties

Name	Type	Short description
Name	Property	The name of the hero
Degree	Property	The degree of the hero
Level	Property	The level of the hero
HP	Property	The health points of the hero
MP	Property	The magic points of the hero
AttackDiff	Property	The difference of the hero's attack points
DefenseDiff	Property	The difference of the hero's defense points
MindDiff	Property	The difference of the hero's mind points
AgilityDiff	Property	The difference of the hero's agility points
MaxHPDiff	Property	The difference of the hero's maximum health points
MaxMPDiff	Property	The difference of the hero's maximum magic points
EXP	Property	The expansion points of the hero

Name

Description

This is the current name of the hero.

Syntax

```
Actor[Index].Name
```

Data type

String

Type

Property

Example

```
$
a[1] = Actor[1].Name
```

At the end a[1] whould contain the name of the first hero (in this case probably "Alex").

Degree

Description

This is the current degree of the hero.

Syntax

```
Actor[Index].Degree
```

Data type

String

Type

Property

Example

```
$
a[1] = Actor[1].Degree
```

At the end a[1] whould contain the degree of the first hero (in this case probably "Soldier").

Level

Description

This is the current level of the hero. If you change this property then the expansion points won't be changed automatically, too.

Syntax

```
Actor[Index].Level
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].Level
```

At the end v[1] whould contain the current level of the first hero (in this case probably 1).

HP

Description

This the current health point value of the hero.

Syntax

```
Actor[Index].HP
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].HP
```

At the end v[1] whould contain the health points of the first hero (in this case probably 48).

MP

Description

This is the current magic point value of the hero.

Syntax

```
Actor[Index].MP
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].MP
```

At the end v[1] whould be the current magic points of the first hero (in this case probably 38).

AttackDiff

Description

This is the difference between the current attack points to the normal attack points of the hero's level. (e. g. if the hero whould usually have 10 attack points on the current level, but totally has 12 attack points, then the difference whould be 2)

Syntax

```
Actor[Index].AttackDiff
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].AttackDiff
```

DefenseDiff

Description

This is the difference between the current defense points to the normal defense points of the hero's level. (For a difference example see AttackDiff)

Syntax

```
Actor[Index].DefenseDiff
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].DefenseDiff
```

MindDiff

Description

This is the difference between the current mind points to the normal mind points of the hero's level. (For a difference example see AttackDiff)

Syntax

```
Actor[Index].MindDiff
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].MindDiff
```

AgilityDiff

Description

This is the difference between the current agility points to the normal agility points of the hero's level. (For a difference example see AttackDiff)

Syntax

```
Actor[Index].AgilityDiff
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].AgilityDiff
```

MaxHPDiff

Description

This is the difference between the current maximum health points to the normal maximum health points of the hero's level. (For a difference example see AttackDiff)

Syntax

```
Actor[Index].MaxHPDiff
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].MaxHPDiff
```

MaxMPDiff

Description

This is the difference between the current maximum magic points to the normal maximum health points of the hero's level. (For a difference example see AttackDiff)

Syntax

```
Actor[Index].MaxMPDiff
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].MaxMPDiff
```

EXP

Description

This is the expansion point value of the hero. If you change this value the level of the hero will not be changed automatically, too.

Syntax

```
Actor[Index].EXP
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Actor[1].EXP
```

9.13 Map object

Description

You can get/set the properties of the current map with the Map object. You can read generic informations (width, height, ...) or change the single chips (upper chip, lower chip).

List of methods/properties

Name	Type	Short description
ID	Property	The id of the map
Width	Property	The width of the map
Height	Property	The height of the map
HeroX	Property	The x coordinate of the hero on the map
HeroY	Property	The y coordinate of the hero on the map
Chipset	Property	The id of the map's used chipset
Lower	Property	The lower chip at a specific position on the map
Upper	Property	The upper chip at a specific position on the map
EventCount	Property	The number of events on the map

ID

Description

This is the id used by the RPG-Maker 2000 to identify the current map.

Syntax

```
Map.ID
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Map.ID
```

At the end v[1] whould contain the Id of the current map (in this case probably 1).

Width

Description

This is the width (in chips) of the current map.

Syntax

```
Map.Width
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Map.Width
```

At the end v[1] whould contain the width of the current map (e. g. on a 20 x 15 sized map this whould be 20).

Height

Description

This is the height (in chips) of the current map.

Syntax

```
Map.Height
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Map.Height
```

At the end v[1] whould contain the height of the current map (e. g. on a 20 x 15 sized map this whould be 15).

HeroX

Description

This is the current hero's x coordinate on the current map.

Syntax

```
Map.HeroX
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Map.HeroX
```

At the end v[1] whould contain the current x coordinate of the hero.

HeroY

Description

This is the current hero's y coordinate on the current map.

Syntax

```
Map.HeroY
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Map.HeroY
```

At the end v[1] whould contain the current y coordinate of the hero.

Chipset

Description

This is the id of the chipset which is used by the current map.

Syntax

```
Map.Chipset
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Map.Chipset
```

At the end v[1] whould contain the chipset id of the current map.

Lower

Description

This is the lower chip at a specific position on the map. A two dimensional index is used to specify the position. The first value is the x coordinate an the second value is the y coordinate. Both values start at 0. The boundaries of the map may not be exceeded.

Syntax

```
Map.Lower[X, Y]
```

Data type

Word

Type

Property

Example

```
$
Map.Lower[0, 0] = 4333
```

At the end the lower chip in the upper left corner of the map (position 0, 0) whould be changed to a poisoned chip (depends on the chipset).

Upper

Description

This is the upper chip at a specific position on the map. A two dimensional index is used to specify the position. The first value is the x coordinate an the second value is the y coordinate. Both values start at 0. The boundaries of the map may not be exceeded.

Syntax

```
Map.Upper[X, Y]
```

Data type

Word

Type

Property

Example

```
$
Map.Upper[Map.Width - 1, 0] = 10000
```

At the end the upper chip in the upper right corner of the map whould be a clear chip (depends on the chipset).

EventCount

Description

This is the total number of events (excluding the hero and the vehicles) on the map. This property can be quite well combined for loops with the MapEvent object.

Syntax

```
Map.EventCount
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Map.EventCount
```

At the end v[1] whould contain the number of events on the current map.

9.14 Event object

Description

You can get/set the properties of an event with the Event object. The Event object requires an index to specify which event shall be responded. This index is the event id used in the RPG-Maker. Because the event id is not continuous numbered this object is not capable for loops. If you want to access events in loops then you should use the MapEvent object. If you want to access a special event (this, hero, boat, ship or airship) you can use the special event constants as index.

List of methods/properties

Name	Type	Short description
ID	Property	The id of the event
MapID	Property	The id of the map where the event is placed
X	Property	The x coordinate of the event
Y	Property	The y coordinate of the event
Dir1	Property	The first part of the event's direction
Dir2	Property	The second part of the event's direction
DirFlags	Property	The direction properties of the event
Exists	Property	The clear state of the event
ScreenX	Property	The x coordinate of the event on the screen
ScreenY	Property	The y coordinate of the event on the screen
Frame	Property	The current frame of the event
Offset	Property	The current offset to the next field of the event
Charset	Property	The name of the event's charset
Frequency	Property	The movement frequency of the event
Speed	Property	The movement speed of the event
Transparency	Property	The transparency of the event
FixDir	Property	The fixed direction property of the event
Phasing	Property	The "walk trough walls" property of the event
StopAnimation	Property	The "no walk animation" property of the event
JumpTime	Property	The jump time value of the event

ID

Description

This is the id used to identify the event in the RPG-Maker.

Syntax

```
Event[Index].ID
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Event[1].ID
```

At the end v[1] whould contain the id of the first event (in this case 1).

MapID

Description

This is the id of the map where the event is currently placed.

Syntax

```
Event[Index].MapID
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Event[1].MapID
```

At the end v[1] whould be the first event's map id (in this case probably 1).

X

Description

This is the x coordinate (in chips) of the event on the current map.

Syntax

```
Event[Index].X
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Event[1].X
```

Y

Description

This is the y coordinate (in chips) of the event on the current map.

Syntax

```
Event[Index].Y
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Event[1].Y
```

Dir1

Description

This is the first part of the direction flags of the event.

Syntax

```
Event[Index].Dir1
```

Data type

Byte

Type

Property

Example

```
$
v[1] = Event[1].Dir1
```

Dir2

Description

This is the second part of the direction flags of the event.

Syntax

```
Event[Index].Dir2
```

Data type

Byte

Type

Property

Example

```
$
Event[HERO].Dir2 = DIR_RIGHT
```

DirFlags

Description

This is the direction property of the event (this is a combination of Dir1 and Dir2).

Syntax

```
Event[Index].DirFlags
```

Data type

Word

Type

Property

Example

```
$
v[1] = Event[1].DirFlags
```

Exists

Description

This property is true whether the event does "exist". If this value is false then the event has been cleared (e. g. with the "clear timer" from the RPG-Maker).

Syntax

```
Event[Index].Exists
```

Data type

Switch

Type

Property

Example

```
$
Event[1].Exists = False
```

At the end the first event whould be cleared.

ScreenX

Description

This is the x coordinate of the event in pixel.

Syntax

```
Event[Index].ScreenX
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Event[1].ScreenX
```

ScreenY

Description

This is the y coordinate of the event in pixel.

Syntax

```
Event[Index].ScreenY
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Event[1].ScreenY
```

Frame

Description

This is the current frame (a piece of the charset) of the event.

Syntax

```
Event[Index].Frame
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Event[1].Frame
```

Offset

Description

This is the offset from the next field of the event in considering to its direction.

Syntax

```
Event[Index].Offset
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Event[1].Offset
```

Charset

Description

This is the name of the event's currently used charset.

Syntax

```
Event[Index].Charset
```

Data type

Dword

Type

Property

Example

```
$
a[1] = Event[1].Charset
```

At the end a[1] whould contain the used charset of the first event (e. g. "CROWN7" for the RTP file "CROWN7.png").

Frequency

Description

This is the current movement frequency of the event. This value should be in the range of 1 to 8.

Syntax

```
Event[Index].Frequency
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Event[1].Frequency
```

Speed

Description

This is the current movement speed of the event. This value should be in the range of 1 to 8.

Syntax

```
Event[Index].Speed
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Event[1].Speed
```

Transparency

Description

This is the current transparency of the event. This value should be in the range of 0 to 8.

Syntax

```
Event[Index].Transparency
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Event[1].Transparency
```

FixDir

Description

This property decides whether the direction of the event is fixed (true means the direction is fixed)

Syntax

```
Event[Index].FixDir
```

Data type

Switch

Type

Property

Example

```
$
Event[THIS].FixDir = True
```

At the end the direction of the current event whould be fixed.

Phasing

Description

This property decides whether an event can walk trough walls, etc. (true means the event can walk through walls).

Syntax

```
Event[Index].Phasing
```

Data type

Switch

Type

Property

Example

```
$
Event[THIS].Phasing = True
```

At the end the current event could walk trough walls.

StopAnimation

Description

This property decides whether an event has a movement animation. (true means the event has no walking animation).

Syntax

```
Event[Index].StopAnimation
```

Data type

Switch

Type

Property

Example

```
$
Event[THIS].StopAnimation = True
```

At the end the current event whouldn't have a walking animation.

JumpTime

Description

This property contains the jump time of the event.

Syntax

```
Event[Index].JumpTime
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Event[1].JumpTime
```

9.15 MapEvent object

Description

You can get/set properties of an event with the MapEvent object. The MapEvent object requires an index which is the number of the event started by zero (e. g. 0 is the first event on the map, 1 is the second, ...). This index is not the same as the event id used in the RPG-Maker to identfy events. If you want to access a single event using its id you should use the Event object. If you want to access a special event (this, hero, boat, ship or airship) you can use the special event constants as index.

List of methods/properties

Given that this object is completely identical to the Event object (except of the different index) see for a list of methods/properties there.

< **Back** 9.14 Event object | 9.15 MapEvent object | **Forward >** 9.16 Picture object

9.16 Picture object

Description

You can get/set the properties of a picture with the Picture object. Additionally you can edit its content. The Picture object requires an index to specify which picture is responded. This index is the picture id.

List of methods/properties

Name	Type	Short description
X	Property	The x coordinate of the picture
Y	Property	The y coordinate of the picture
Width	Property	The width of the picture
Height	Property	The height of the picture
Magnification	Property	The magnification of the picture
Transparency	Property	The transparency of the picture
Red	Property	The red coloration of the picture
Green	Property	The green coloration of the picture
Blue	Property	The blue coloration of the picture
Chroma	Property	The chroma of the picture
Action	Property	The action of the picture
ActionStrength	Property	The action strength of the picture
ActionValue	Property	The current action value of the picture
MapMove	Property	The move with map property of the picture
MapX	Property	The relative x coordinate of the picture on the map
MapY	Property	The relative y coordinate of the picture on the map
Pixel	Property	A pixel of the picture at a specific position
Palette	Property	A specified palette entry
UseMaskColor	Property	The "use mask color" property of the picture
DrawLine	Method	Draws a line into the picture
FillRect	Method	Draws a filles rectangular into the picture
CopyRect	Method	Copies a rectangular from one picture into an other
BltRect	Method	Copies a rectangular from on picture into an

		other an skips a specific color
FlushPalette	Method	Applies changes of the palette

X

Description

This is the current x coordinate of the picture (in pixel). The RPG-Maker 2000 relates this coordinate to the center of the picture. If this value is too huge then the RPG_RT could crash. A more specific description can be found at the known bugs.

Syntax

```
Picture[Index].X
```

Data type

Double

Type

Property

Example

```
$
f[1] = Picture[1].X
```

At the end f[1] whould contain the x coordinate of the first picture (e. g. at the position 160:120 this whould be 160).

Y

Description

This is the current y coordinate of the picture (in pixel). The RPG-Maker 2000 relates this coordinate to the center of the picture. If this value is too huge then the RPG_RT could crash. A more specific description can be found at the known bugs.

Syntax

```
Picture[Index].Y
```

Data type

Double

Type

Property

Example

```
$
f[1] = Picture[1].Y
```

At the end f[1] whould contain the y coordinate of the first picture (e. g. at the position 160:120 this whould be 120).

Width

Description

This is the width of the picture.

Syntax

```
Picture[Index].Width
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Picture[1].Width
```

Height

Description

This is the height of the picture.

Syntax

```
Picture[Index].Height
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Picture[1].Height
```

Magnification

Description

This is the magnification of the picture. This value is in percent (e. g. 100 equals to 100% what whould be the normal size). This value should be in the range of 1 to 2000.

Syntax

```
Picture[Index].Magnification
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Picture[1].Magnification
```

Transparency

Description

This is the transparency of the picture. This value is in percent (e. g. 0 means that the picture is not transparent). This value should be in the range of 0 to 100.

Syntax

```
Picture[Index].Transparency
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Picture[1].Transparency
```

Red

Description

This is the red coloration of the picture. This value is in percent (e. g. 100 is the default view). This value should be in the range of 0 to 200.

Syntax

```
Picture[Index].Red
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Picture[1].Red
```

Green

Description

This is the green coloration of the picture. This value is in percent (e. g. 100 is the default view). This value should be in the range of 0 to 200.

Syntax

```
Picture[Index].Green
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Picture[1].Green
```

Blue

Description

This is the blue coloration of the picture. This value is in percent (e. g. 100 is the default view). This value should be in the range of 0 to 200.

Syntax

```
Picture[Index].Blue
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Picture[1].Blue
```

Chroma

Description

This is the chrominance of the picture. This value is in percent (e. g. 100 is the default view). This value should be in the range of 0 to 200.

Syntax

```
Picture[Index].Chroma
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Picture[1].Chroma
```

Action

Description

This is the action of the picture. You can use the action constants for this.

Syntax

```
Picture[Index].Action
```

Data type

Dword

Type

Property

Range

0 to 2

Example

```
$
v[1] = Picture[1].Action
```

ActionStrength

Description

This is the action strength of the value of the picture.

Syntax

```
Picture[Index].ActionStrength
```

Data type

Dword

Type

Property

Range

-10 to 10

Example

```
$
v[1] = Picture[1].ActionStrength
```

ActionValue

Description

This is the current action value of the picture.

Syntax

```
Picture[Index].ActionValue
```

Data type

Double

Type

Property

Example

```
$
f[1] = Picture[1].ActionValue
```

At the end f[1] whould contain the action value (if the action whould be rotation then this whould be the angle in RPG format) of the first picture.

MapMove

Description

If this property is true then the move with map property of the picture is used.

Syntax

```
Picture[Index].MapMove
```

Data type

Switch

Type

Property

Example

```
$
Picture[1].MapMove = True
```

At the end the move with map property of the first picture whould be activated (the picture whould scroll with the map).

MapX

Description

This is x coordinate used in the show picture command for pictures with activated move with map option.

Syntax

```
Picture[Index].MapX
```

Data type

Double

Type

Property

Example

```
$
f[1] = Picture[1].MapX
```

MapY

Description

This is the y coordinate used in the show picture command for pictures with activated move with map option.

Syntax

```
Picture[Index].MapY
```

Data type

Double

Type

Property

Example

```
$
f[1] = Picture[1].MapY
```

Pixel

Description

You cann access pixels with this property. This property requires a two dimensional index. The first value is the x coordinate and the second value is the y coordinate. The first pixel is at [0, 0] and the last pixel is at [width - 1, height - 1]. The boundaries may not be exceeded. The value of a pixel is the index of the used palette entry (0 to 255).

Syntax

```
Picture[Index].Pixel[X, Y]
```

Data type

Byte

Type

Property

Example

```
$
v[1] = Picture[1].Pixel[0, 0]
```

At the end v[1] whould be the index of the palette entry used for the pixel in the upper left corner of the first picture.

Palette

Description

You can access the color values used in a picture's palette with this property. This option requires an index which responded to the palette entry in the range of 0 to 255. If you change one or more palette entries you must call FlushPalette to apply the changes.

Syntax

```
Picture[Index].Palette[Color]
```

Data type

Dword

Type

Property

Example

```
$
v[1] = Picture[1].Palette[0]
```

At the end v[1] whould contain the RGB color of the first palette entry of the first picture.

UseMaskColor

Description

If this property is true then the mask color will not be drawn.

Syntax

```
Picture[Index].UseMaskColor
```

Data type

Switch

Type

Property

Example

```
$
Picture[1].UseMaskColor = True
```

At the end the mask color of the first picture whouldn't be drawn.

DrawLine

Description

With this option you can draw a one pixel thick line. The start coordinates are X1 and Y1. The end coordinates are (these are not always reached) are X2 and Y2. The color for this line is the index for the palette entry.

Syntax

```
Picture[Index].DrawLine(X1, Y1, X2, Y2,
Color)
```

Return value

None

Type

Method

Parameter: X1

Description

The x coordinate where the line begins.

Data type

Dword

Parameter: Y1

Description

The y coordinate where the line begins.

Data type

Dword

Parameter: X2

Description

The x coordinate where the line ends.

Data type

Dword

Parameter: Y2

Description

The y coordinate where the line ends.

Data type

Dword

Parameter: Color

Description

The used palette entry for the color of the line.

Data type

Byte

Example

```
$
Picture[1].DrawLine(0, 0, 24, 24, 1)
```

At the end there whould be a line drawn from 0, 0 to 24, 24 in the color of the second palette entry (= index 1).

FillRect

Description

You can draw a filled rectangle with this method. The start coordinates are left and top. The end coordinates (these are never reached) are right and bottom. The color for this rectangle is the index for the palette entry.

Syntax

```
Picture[Index].FillRect(Left, Top, Right,
Bottom, Color)
```

Return value

None

Type

Method

Parameter: Left

Description

The left border of the rectangle.

Data type

Dword

Parameter: Top

Description

The upper border of the rectangle.

Data type

Dword

Parameter: Right

Description

The right border + 1 of the rectangle (= left + width).

Data type

Dword

Parameter: Bottom

Description

The lower border + 1 of the rectangle (= top + height).

Data type

Dword

Parameter: Color

Description

The used palette entry for the color of the rectangle.

Data type

Byte

Example

```
$
Picture[1].FillRect(0, 0, 24, 24, 1)
```

At the end there whould be a rectangle drawn from 0, 0 to (including) 23, 23 in the color of the second palette entry (= index 1).

CopyRect

Description

With this method you can copy a rectangular area from one picture into an other. The palette entries will not be adjusted. So if the source picture has the color red as palette entry 0 and the destination picture has the color green as palette entry 0 then all red pixels (that refer to palette entry 0) will be green in the destination picture. X and Y specify the position where the area should be drawn in the current picture. The source id must be a valid picture id of an other (!) picture. The parameters Left, Top, Right and Bottom describe the range and the coordinates of the area in the source picture.

Syntax

```
Picture[Index].CopyRect(X, Y,
Source-ID, Left, Top, Right, Bottom)
```

Return value

None

Type

Method

Parameter: X

Description

The x coordinate where the area shall be drawn to.

Data type

Dword

Parameter: Y

Description

The y coordinate where the area shall be drawn to.

Data type

Dword

Parameter: Source-ID

Description

The id of the source picture. This value may not the same as the destination picture.

Data type

Dword

Parameter: Left

Description

The left border of the area in the source picture.

Data type

Dword

Parameter: Top

Description

The upper border of the area in the source picture.

Data type

Dword

Parameter: Right

Description

The right border + 1 of the area in the source picture (= left + width).

Data type

Dword

Parameter: Bottom

Description

The lower border + 1 of the area in the source picture (= top + height).

Data type

Dword

Example

```
$
Picture[1].CopyRect(0, 0, 2, 0, 0, 24, 24)
```

At the end there whould be a rectangular area copied from the second picture to the first which is at 0, 0 and has a size of 24 x 24 pixels.

BltRect

Description

With this method you can copy a rectangular area from one picture into an other, but the specified mask color will be skipped. The palette entries will not be adjusted. So if the source picture has the color red as palette entry 0 and the destination picture has the color green as palette entry 0 then all red pixels (that refer to palette entry 0) will be green in the destination picture. X and Y specify the position where the area should be drawn in the current picture. The source id must be a valid picture id of an other (!) picture. The parameters Left, Top, Right and Bottom describe the range and the coordinates of the area in the source picture.

Syntax

```
Picture[Index].CopyRect(X, Y,
Source-ID, Left, Top, Right, Bottom,
MaskColor)
```

Return value

None

Type

Method

Parameter: X

Description

The x coordinate where the area shall be drawn to.

Data type

Dword

Parameter: Y

Description

The y coordinate where the area shall be drawn to.

Data type

Dword

Parameter: Source-ID

Description

The id of the source picture. This value may not the same as the destination picture.

Data type

Dword

Parameter: Left

Description

The left border of the area in the source picture.

Data type

Dword

Parameter: Top

Description

The upper border of the area in the source picture.

Data type

Dword

Parameter: Right

Description

The right border + 1 of the area in the source picture (= left + width).

Data type

Dword

Parameter: Bottom

Description

The lower border + 1 of the area in the source picture (= top + height).

Data type

Dword

Parameter: MaskColor

Description

Specifies the palette entry whose pixels shall not be copied.

Data type

Byte

Example

```
$
Picture[1].BltRect(0, 0, 2, 0, 0, 24, 24, 0)
```

At the end there whould be a rectangular area copied from the second picture to the first skipping the color 0. The area is drawn at 0, 0 and has a size of 24 x 24 pixels.

FlushPalette

Description

With this method you can apply changes of a palette.

Syntax

```
Picture[Index].FlushPalette()
```

Return value

None

Type

Method

Example

```
$
Picture[1].Palette[0] = 0xFF00FF;
Picture[1].FlushPalette()
```

At the end the color of the first palette entry whould have been changed to magenta (Color code #FF00FF) and used to draw the picture.

9.17 Client object

Description

You can establish a connection to other computers via the TCP/IP protocol with the Client object. If a connection has been established successfully then you call it a socket. Using such a socket you can send/receive data. In DestinyScript there are two kinds of sockets (= socket types): sockets using the Destiny protocol (= DestinySockets) and such who don't have a specific protocol (= RAW sockets). Sockets that are not longer required should be closed. The Client object requires an index to specify the responded socket. This index is in the range of 0 to 31. The assignment of the index can be done with the Server object if you accept incoming connections.

List of methods/properties

Name	Type	Short description
Type	Property	The used socket type
State	Property	The current connection state
LocalIP	Property	The used ip of the own computer
LocalPort	Property	The used port of the own computer
RemoteIP	Property	The used ip of the other computer
RemotePort	Property	The used port of the other computer
Connect	Method	Establishes a connection to an other computer
Close	Method	Closes an open connection
SendVariable	Method	Sends a variable over a DestinySocket
SendByte	Method	Sends a byte over a DestinySocket
SendWord	Method	Sends a word over a DestinySocket
SendDword	Method	Sends a dword over a DestinySocket
SendDouble	Method	Sends a double over a DestinySocket
SendString	Method	Sends a string over a DestinySocket
SendSwitch	Method	Sends a switch over a DestinySocket
SendRawData	Method	Sends data over a RAW socket
GetRecvType	Method	Returns the kind of received data from a DestinySocket
GetRecvLength	Method	Returns the number of bytes received on a socket
RecvID	Method	Receives the id from the current data package of a DestinySocket
RecvVariable	Method	Receives a variable from a DestinySocket
RecvByte	Method	Receives a byte from a DestinySocket
RecvWord	Method	Receives a word from a DestinySocket
RecvDword	Method	Receives a dword from a DestinySocket

RecvDouble	Method	Receives a double from a DestinySocket
RecvString	Method	Receives a string from a DestinySocket
RecvSwitch	Method	Receives a switch from a DestinySocket
RecvRawData	Method	Receives data from a RAW socket

Type

Description

This property specifies whether DestinySocket (= 0) or a RAW socket (= 1) is used. You can use the socket type constants for this.

Syntax

```
Client[Index].Type
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Client[0].Type
```

State

Description

This property represents the current state of the socket. You can use the socket state constants for this.

Syntax

```
Client[Index].State
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Client[0].State
```

LocalIP

Description

The value of this property is the ip used on the own computer for the connection.

Syntax

```
Client[Index].LocalIP
```

Data type

String

Type

Property, read-only

Example

```
$
a[1] = Client[0].LocalIP
```

LocalPort

Description

The value of this property is the port used on the own computer for the connection.

Syntax

```
Client[Index].LocalPort
```

Data type

Dword

Type

Property, read-only

Range

1 to 65535

Example

```
$
v[1] = Client[0].LocalPort
```

RemoteIP

Description

The value of this property is the ip used on the other computer for the connection.

Syntax

```
Client[Index].RemoteIP
```

Data type

String

Type

Property, read-only

Example

```
$
a[1] = Client[0].RemoteIP
```

RemotePort

Description

The value of this property is the port used on the other computer for the connection.

Syntax

```
Client[Index].RemotePort
```

Data type

Dword

Type

Property, read-only

Range

1 to 65535

Example

```
$
v[1] = Client[0].RemotePort
```

Connect

Description

You can establish a connection with this method using the the TCP/IP protocol. If you call this method you define whether it uses a DestinySocket or a RAW socket. Because the windows function used to establish a connection is a "blocking call" the DestinyScript (and even the game) will freeze until a connection is established or the timeout occured. The timeout value is ca. 2 seconds. If a connection has been established or not can be checked with the state property of this object.

Syntax

```
Client[Index].Connect(Address, Port, Sockettype)
```

Return value

None

Type

Method

Parameter: Address

Description

The address of the destination computer. This can either be an ip (e. g. "192.168.1.1") or a hostname (e. g. "bananen-joe.de").

Data type

String

Parameter: Port

Description

The port of the destination computer. (e. g. Port 80 for http)

Data type

Dword

Range

1 to 65535

Parameter: Sockettype

Description

The socket type for the connection. This can be either DestinySocket (= 0) or RAW socket (= 1). You can use the socket constants for this.

Data type

Dword

Range

0 to 1

Example

```
$
Client[0].Connect("bananen-joe.de", 80,
```

```
        SOCK_RAW)
```

At the end a connection via internet whould be established to the webserver "bananen-joe.de".

Close

Description

With this method an open connection can be closed. If a socket is closed it can be used to open a new connection. If you close a already closed socket this doesn't raise any problems.

Syntax

```
Client[Index].Close()
```

Return value

None

Type

Method

Example

```
$
Client[0].Close()
```

SendVariable

Description

With this method you can send a variable over a connected DestinySocket. This method can't be used with RAW sockets.

Syntax

```
Client[Index].SendVariable(ID, Variable)
```

Return value

None

Type

Method

Parameter: ID

Description

The index of the variable. This value may differ from the real variable index.

Data type

Dword

Parameter: Variable

Description

The value of the variable.

Data type

Dword

Example

```
$
Client[0].SendVariable(1, v[1])
```

At the end the value of the first variable whould be send over the first socket (which is a connected DestinySocket).

SendByte

Description

With this method you can send a byte over a connected DestinySocket. This method can't be used with RAW sockets.

Syntax

```
Client[Index].SendByte(ID, Byte)
```

Return value

None

Type

Method

Parameter: ID

Description

The associated index of the byte.

Data type

Dword

Parameter: Byte

Description

The value of the byte.

Data type

Byte

Example

```
$
Client[0].SendByte(1, v[1])
```

At the end the value of the first variable whould be send (as byte) over the first socket (which is a connected DestinySocket).

SendWord

Description

With this method you can send a word over a connected DestinySocket.
This method can't be used with RAW sockets.

Syntax

```
Client[Index].SendWord(ID, Word)
```

Return value

None

Type

Method

Parameter: ID

Description

The associated index of the word.

Data type

Dword

Parameter: Word

Description

The value of the word.

Data type

Word

Example

```
$
Client[0].SendWord(1, v[1])
```

At the end the value of the first variable whould be send (as dword) over the first socket (which is a connected DestinySocket).

SendDword

Description

With this method you can send a dword over a connected DestinySocket.
This method can't be used with RAW sockets.

Syntax

```
Client[Index].SendDword(ID, Dword)
```

Return value

None

Type

Method

Parameter: ID

Description

The associated index of the dword.

Data type

Dword

Parameter: Dword

Description

The value of the dword.

Data type

Dword

Example

```
$
Client[0].SendDword(1, d[1])
```

At the end the value of the first dword whould be send over the first socket (which is a connected DestinySocket).

SendDouble

Description

With this method you can send a double over a connected DestinySocket.
This method can't be used with RAW sockets.

Syntax

```
Client[Index].SendDouble(ID, Double)
```

Return value

None

Type

Method

Parameter: ID

Description

The associated index of the double.

Data type

Dword

Parameter: Double

Description

The value of the double.

Data type

Double

Example

```
$
Client[0].SendDouble(1, f[1])
```

At the end the value of the first double whould be send over the first socket (which is a connected DestinySocket).

SendString

Description

With this method you can send a string over a connected DestinySocket. The string may not exceed a length of 255 chars. This method can't be used with RAW sockets.

Syntax

```
Client[Index].SendString(ID, String)
```

Return value

None

Type

Method

Parameter: ID

Description

The associated index of the string.

Data type

Dword

Parameter: String

Description

The value of the string. This value may not exceed 255 chars!

Data type

String

Example

```
$
Client[0].SendString(1, a[1])
```

At the end the value of the first string whould be send over the first socket (which is a connected DestinySocket).

SendSwitch

Description

With this method you can send a switch over a connected DestinySocket. This method can't be used with RAW sockets.

Syntax

```
Client[Index].SendSwitch(ID, Switch)
```

Return value

None

Type

Method

Parameter: ID

Description

The associated index of the switch.

Data type

Dword

Parameter: Switch

Description

The value of the switch.

Data type

Switch

Example

```
$
Client[0].SendSwitch(1, s[1])
```

At the end the value of the first string whould be send over the first socket (which is a connected DestinySocket).

SendRawData

Description

With this method you can send a specified amount of data over a RAW socket. The data is processed binary, so it is possible to send less data than the value contains (e. g. you can send 4 bytes of a double although it is usually 8 bytes long). This method can't be used with DestinySockets.

Syntax

```
Client[Index].SendRawData(Datasource,
Length)
```

Return value

None

Type

Method

Parameter: Datasource

Description

The data source where the bytes are taken from.

Data type

Alle

Parameter: Length

Description

The number of bytes being send. This may not exceed the length of the data source. For example you can't send 9 bytes from a double (which has a maximum length of 8 bytes).

Data type

Dword

Example

```
$
a[1] = "GET / HTTP/1.0" + CRLF +
"Host: bananen-joe.de" + CRLF + CRLF;
Client[0].SendRawData(a[1], String.Length(a[1]))
```

At the end content of a[1] (which is a http request header) whould be send over a RAW socket (compare the example of connect) to the webserver "bananen-joe.de".

GetRecvType

Description

With this method you can determine whether a data packet has been received completely. If the return value is negative value then the data paket isn't complete (the return value is the negative data type), otherwise (if the return value is positive) the data paket is complete and the return value tells you what kind it is. You can use the data type constants for this. If there is no data paket this method will return 0. This method can only be used with DestinySockets. If you call this method the Destiny.dll will receive data on this socket if possible.

Syntax

```
Client[Index].GetRecvType()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = Client[0].GetRecvType()
```

Am Ende wäre v[1] der Data type des zuletzt empfangenen Datenpakets bei einem verbundenen DestinySocket. Wenn Examplesweise der andere Computer mit SendVariable eine Variable versendet hätte, würde hier die Zahl 1 (= TYPE_VARIABLE) zurückgegeben werden.

GetRecvLength

Description

Returns the number of received data (in bytes). This value can't be more than 500 bytes due to a weakness of the Destiny.dll (the internal buffer is limited to 500 bytes). This method can be used with each socket type. If you call this method the Destiny.dll will receive data on this socket if possible.

Syntax

```
Client[Index].GetRecvLength()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = Client[0].GetRecvLength()
```

RecvID

Description

Receives the associated id of the current data package. This method must be called implicitly before other recv methods are called on DestinySockets (except for GetRecvType and GetRecvLength) because these methods remove the current data package from the internal buffer. RecvID doesn't remove the current data package from the internal buffer. This method can only be used with DestinySockets.

Syntax

```
Client[Index].RecvID()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = Client[0].RecvID()
```

At the end v[1] whould contain the id of the last received data package on a connected DestinySocket. For example if the other computer executes SendVariable(1, 2) this method (on our computer) whould return 1.

RecvVariable

Description

Receives the variable value of the current data package and removes that data package from the internal buffer. This method can only be used with DestinySockets.

Syntax

```
Client[Index].RecvVariable()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = Client[0].RecvVariable()
```

At the end v[1] whould contain the variable value of the last received data package on a connected DestinySocket. For example if the other computer executes SendVariable(1, 2) this method (on our computer) whould return 2.

RecvByte

Description

Receives the byte value of the current data package and removes that data package from the internal buffer. This method can only be used with DestinySockets.

Syntax

```
Client[Index].RecvByte()
```

Return value

Byte

Type

Method

Example

```
$
v[1] = Client[0].RecvByte()
```

RecvWord

Description

Receives the word value of the current data package and removes that data package from the internal buffer. This method can only be used with DestinySockets.

Syntax

```
Client[Index].RecvWord()
```

Return value

Word

Type

Method

Example

```
$
v[1] = Client[0].RecvWord()
```

RecvDword

Description

Receives the dword value of the current data package and removes that data package from the internal buffer. This method can only be used with DestinySockets.

Syntax

```
Client[Index].RecvDword()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = Client[0].RecvDword()
```

RecvDouble

Description

Receives the double value of the current data package and removes that data package from the internal buffer. This method can only be used with DestinySockets.

Syntax

```
Client[Index].RecvDouble()
```

Return value

Double

Type

Method

Example

```
$
f[1] = Client[0].RecvDouble()
```

RecvString

Description

Receives the string value of the current data package and removes that data package from the internal buffer. This method can only be used with DestinySockets.

Syntax

```
Client[Index].RecvString()
```

Return value

String

Type

Method

Example

```
$
a[1] = Client[0].RecvString()
```

RecvSwitch

Description

Receives the switch value of the current data package and removes that data package from the internal buffer. This method can only be used with DestinySockets.

Syntax

```
Client[Index].RecvSwitch()
```

Return value

Switch

Type

Method

Example

```
$
a[1] = Client[0].RecvSwitch()
```

RecvRawData

Description

Receives a specified amount of bytes and returns this value as a specified data type. If the data type has a minimum length (e. g. double requires 8 bytes total) the missing bytes will be filled with zeros. Finally the received amount of bytes will be removed from the internal buffer. The amount of bytes may not exceed 500 bytes. This method can only be used with RAW sockets.

Syntax

```
Client[Index].RecvRawData(DataType, Length)
```

Return value

Depends on the parameter DataType.

Type

Method

Parameter: DataType

Description

Defines the data type used for the return value. You can use the data type constants for this.

Data type

Dword

Range

1 to 7

Parameter: Length

Description

Defines the number of bytes to receive. If this value is too small for the specified data type then the missing bytes will be filled with zero bytes.

Data type

Dword

Range

1 to 500

Example

```
$
v[1] = Client[0].GetRecvLength();
a[1] = Client[0].RecvRawData(TYPE_STRING, v[1])
```

At the end a[1] whould contain a string in the length of the received bytes of the connected RAW socket. Usally there must be checked whether more than 0 bytes are received before the call of RecvRawData.

9.18 Server object

Description

You can accept incoming connections from other computers via the TCP/IP protocol with the Server object. Accepted connections can be accessed via the Client object.

List of methods/properties

Name	Type	Short description
Type	Property	The used socket type
State	Property	The current connection state
Listen	Method	Waits for incoming connections
Close	Method	Stops the waiting for incoming connections
Accept	Method	Accepts an incoming connection

Type

Description

This property specifies whether accepted connections will be DestinySockets (= 0) or RAW-Sockets (= 1). You can use the socket type constants for this. This value will be specified with the call of the listen method.

Syntax

```
Server.Type
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Server.Type
```

State

Description

This property specifies the current state of the server socket. You can use the socket state constants for this.

Syntax

```
Server.State
```

Data type

Dword

Type

Property, read-only

Example

```
$
v[1] = Server].State
```

Listen

Description

With this method you can set the server socket into the listening state. If a server socket is in listening state then it can accept incoming connections. To accept an incoming connection you can use the accept method. On some computers the firewall can make trouble. In this case it is impossible to accept incoming connections. Fore more specific information see at the known bugs.

Syntax

```
Server.Listen(Port, SocketType)
```

Return value

None

Type

Method

Parameter: Port

Description

The local port used for incoming connections. The clients must connect to this port if they want to establish a connection.

Data type

Dword

Range

1 to 65535

Parameter: SocketType

Description

The socket types of the client sockets that are created using the accept method. You can use the socket type constants for this.

Data type

Dword

Range

0 to 1

Example

```
$
Server.Listen(12345, SOCK_DESTINY)
```

At the end incoming connections from the network (or internet) whould be able to accept.

Close

Description

With this method you can close the server socket. In this case no more incoming connections will be accepted. Already connected clients are still connected.

Syntax

```
Server.Close()
```

Return value

None

Type

Method

Example

```
$
Server.Close()
```

Accept

Description

With this method you can accept an incoming connection (if there is one). The accepted connection will be dedicated to a Client object. The return value of this method is the index used for the client object that has accepted the connection. If there wasn't an incoming connection pending then this method will return -1. You can specify a client with the parameter client. If you do so then only this client object will be used to accept a connection and the client socket will be closed if necessary. If the next free socket shall be used then you can use the NEXT_FREE_SOCKET constant (= -1).

Syntax

```
Server.Accept(Client)
```

Return value

Dword

Type

Method

Parameter: Client

Description

The index of the client object which shall accept the incoming connection. For the next free client object use the NEXT_FREE_SOCKET constant.

Data type

Dword

Range

-1 to 31

Example

```
$
v[1] = Server.Accept(NEXT_FREE_SOCKET)
```

At the end v[1] whould contain the index of the client object which has accepted the incoming connection (if there was one). This only works if the server socket was in listening state. If a client has established a connection (and it has been accepted) then you can access this connection (in this example) with Client[v[1]] like a normal client socket.

< Back		**Forward >**
9.17 Client object	9.18 Server object	9.19 File object

9.19 File object

Description

You can read/write files with the File object. Due to security reasons each file access is only possible inside of the game directory. The File object requires an index in the range of 0 to 31. This index is required to specify which file stream shall be used (in other words you can open multiple files at the same time).

List of methods/properties

Name	Type	Short description
Open	Method	Opens a file for reading and/or writing
Close	Method	Closes a previously opened file
ReadRawData	Method	Reads data from the file
WriteRawData	Method	Writes data into the file
GetFilePointer	Method	Returns the current read/write position of the file
SetFilePointer	Method	Sets the current read/write position of the file
Length	Method	Returns the current length of the file
Truncate	Method	Truncates the rear part of the file

Open

Description

With this method you can open a file for reading/writing. If you open a file for writing and it doesn't exist then it will be created automatically. You can specify only a relative path to the game directory as file name. The parameter file mode specifies either the file shall be opened for reading and/or writing. You can use the file mode constants for this. After opening and reading/writing all required data you must close the file with the close method to ensure that the file handles are free again.

Syntax

```
File[Index].Open(Filename, FileMode)
```

Return value

None

Type

Method

Parameter: Filename

Description

Specifies the filename (including the relative path if necessary) of the file which shall be opened. Files may only be opened inside the game directory (and subdirectories)!

Data type

String

Parameter: FileMode

Description

Specifies whether a file shall be opened for reading/writing. To specify more than one file mode you must use the binary or operator (e. g. FILE_READ | FILE_WRITE). If you open a file for appending it will always be opened with write access. You can use the file mode constants for this parameter.

Data type

Dword

Range

1 to 7

Example

```
$
File[0].Open("Textfile.txt", FILE_WRITE)
```

At the end a file with the name "Textfile.txt" whould be opened in the game directory for writing. If this file whould not exist then it whould be created.

Close

Description

With this method you can close a previously opened file. The file handle will be free again and each writing operation will be finally done. (On some operating systems it is possible that some write operations are only done after you closed the file! Hence you should close the file if you no longer need it.)

Syntax

```
File[Index].Close()
```

Return value

None

Type

Method

Example

```
$
File[0].Close()
```

At the end the file previously opened with the open method whould be closed. Hence you can open a file with the File object (using index 0) again.

ReadRawData

Description

With this method you can read an amount of bytes from a previously opened file (with read access). The first parameter specifies the data type which is used to return the bytes read. You can use the data type constants for this. If you read data from a text file then you should only use strings for reading/writing. The second parameter specifies the amount of bytes which is being read. This value may not exceed the maximum size of the data type (e. g. 8 bytes for double, 4 bytes for dword, ...). If this value is smaller than the maximum size of the specified data type then the missing bytes will be filled with zero-bytes to reach the required length. If you specify (for each data type expect string) a length of 0 bytes then the required number of bytes will be read automatically. After reading the bytes the internal file pointer will be increased automatically by the amount of bytes.

Syntax

```
File[Index].ReadRawData(DataType, Length)
```

Return value

Depends on the parameter DataType.

Type

Method

Parameter: Data type

Description

Specifies the data type of the return value. You can use the data type constants for this.

Data type

Dword

Range

1 to 7

Parameter: Length

Description

Specifies the amount of bytes that will be read. If this value is too small for the specified data type then the data type will be filles with zero-bytes. You can specify a 0 here for each data type expect string to detect automatically which length is required.

Data type

Dword

Example

```
$
a[1] = File[0].ReadRawData(TYPE_STRING, File[0].Length()))
```

At the end the entire content of a previously opened (text-)file whould be read into a[1].

WriteRawData

Description

With this method you can write data into a previously opened file (with write access). The first parameter specifies the data source (which contents the bytes to be written). If the target file is a text file then you should only write strings. The second parameter specifies the amount of bytes that shall be written into the file. This value may not exceed the maximum length of the data type used by the data source. You can specify a 0 here to write the entire data source (unlike ReadRawData this works with strings) into the file. After writing the data into the file the internal file pointer will be increased automatically by the amount of bytes written. If you write past the end of file then the file size will increase automatically.

Syntax

```
File[Index].WriteRawData(DataSource, Length)
```

Return value

None

Type

Method

Parameter: DataSource

Description

Specifies the data source which contains the bytes that shall be written into the file.

Data type

All

Parameter: Length

Description

Specifies the amount of bytes that shall be written into the file. If this value is 0 then the entire content of the data source will be written into the file automatically.

Data type

Dword

Example

```
$
File[0].WriteRawData(v[1], 0)
```

At the end the entire content of the first variable whould be written into a (binary) file.

GetFilePointer

Description

With this method you can retrieve the position of the internal file pointer.

Syntax

```
File[Index].GetFilePointer()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = File[0].GetFilePointer()
```

SetFilePointer

Description

With this method you can set the new position of the internal file pointer.

Syntax

```
File[Index].SetFilePointer(NewFilePointer)
```

Return value

None

Type

Method

Parameter: NewFilePointer

Description

Specifies the new position of the file pointer. This value may not exceed the current file length.

Data type

Dword

Example

```
$
File[0].SetFilePointer(0)
```

At the end the internal file pointer whould point to the begin (= 0) of the file.

Length

Description

With this method you can retrieve the current length of a file.

Syntax

```
File[Index].Length()
```

Return value

Dword

Type

Method

Example

```
$
v[1] = File[0].Length()
```

Truncate

Description

With this method you can truncate the rear part of a file. The file will be truncated after the position of the internal file pointer. For example you can clear to entire content of a file if the internal file pointer points to the begin of the file (= 0). To truncate a file you must open it with write access.

Syntax

```
File[Index].Truncate()
```

Return value

None

Type

Method

Example

```
$
File[0].SetFilePointer(0);
File[0].Truncate()
```

At the end the entire file whould be cleared.

9.20 Directory object

Description

You can edit directory and file structures with the Directory object. Additionally you can browse directories. Like at the File object you can only access the game directory and its subdirectories.

List of methods/properties

Name	Type	Short description
CreateDir	Method	Creates a directory
RemoveDir	Method	Removes an empty(!) directory
Rename	Method	Renames a file/directory or moves it
CopyFile	Method	Copies a file
DeleteFile	Method	Deletes a file
GetAttributes	Method	Returns the attributes of a file/directory
SetAttributes	Method	Sets the attributes of a file/directory
FindFirst	Method	Starts to browse a directory
FindNext	Method	Continues the browsing of a directory
FindClose	Method	Stops the browsing of a directory

CreateDir

Description

With this method you can create a new directory. The name of the directory has a relative path to the game directory.

Syntax

```
Directory.CreateDir(Directory)
```

Return value

None

Type

Method

Parameter: Directory

Description

Specifies the name (inclusive the relative path if necessary) of the new directory. You can only create directories inside of the game directory (and subdirectories).

Data type

String

Example

```
$
Directory.CreateDir("Picture\Content")
```

At the end a new subdirectory whould be created with the name "Content" in the Picture folder.

RemoveDir

Description

With this method you can remove an empty directory. The name of the directory has a relative path to the game directory.

Syntax

```
Directory.RemoveDir(Directory)
```

Return value

None

Type

Method

Parameter: Directory

Description

Specifies the name (inclusive the relative path if necessary) of the directory that shall be removed. You can only remove directories inside of the game directory (and subdirectories).

Data type

String

Example

```
$
Directory.RemoveDir("Testfolder")
```

At the end the directory with the name "Testfolder"e; (which is inside of the game directory) whould be removed.

Rename

Description

With this method you can rename or move files and directories. A file/directory is moved if it gets a new path (without a new name). Otherwise it will be renamed (and moved if the path is different).

Syntax

```
Directory.Rename(OldPath, NewPath)
```

Return value

None

Type

Method

Parameter: OldPath

Description

Specifies the current (relative) path of the file/directory. You can only rename files/directories inside of the game directory (and subdirectories).

Data type

String

Parameter: NewPath

Description

Specifies the new (relative) path of the file/directory. You can only rename files/directories inside of the game directory (and subdirectories).

Data type

String

Example

```
1 $
2 Directory.Rename("Folder old\File.txt",
3 "Folder new\File.txt);
4 Directory.Rename("Folder old\Subfolder",
5 "Folder old\Subfolder2)
```

At the end the file "File.txt" whould be moved from the directory "Folder old" into the directory "Folder new" (Line 2 & 3). Additionally the directory "Subfolder" whould be renamed to "Subfolder2" (Line 4 & 5).

CopyFile

Description

With this method you can copy files.

Syntax

```
Directory.CopyFile(SourceFile, DestinationFile)
```

Return value

None

Type

Method

Parameter: SourceFile

Description

Specifies the (relative) path of the file which shall be copied. You can only copy files inside of the game directory (and subdirectories).

Data type

String

Parameter: DestinationFile

Description

Specifies the (relative) path of the copy. You can only copy files inside of the game directory (and subdirectories).

Data type

String

Example

```
$
Directory.CopyFile("RPG_RT.exe", "RPG_RT2.exe")
```

At the end the file "RPG_RT.exe" whould be copied as "RPG_RT2.exe".

DeleteFile

Description

With this method you can delete files.

Syntax

```
Directory.DeleteFile(File)
```

Return value

None

Type

Method

Parameter: File

Description

Specifies the (relative) path of the file that shall be deleted. You can only delete files inside of the game directory (and subdirectories).

Data type

String

Example

```
$
Directory.DeleteFile("Test.txt")
```

At the end the file "Test.txt" whould be deleted.

GetAttributes

Description

With this method you can retrieve the attributes of a file/directory. The return value is a combination of the file attribute constants.

Syntax

```
Directory.GetAttributes(Path)
```

Return value

Dword

Type

Method

Parameter: Path

Description

Specifies the (relative) path of the file/directory whose attributes shall be retrieved. You can only access the attributes of files/directories inside of the game directory (and subdirectories).

Data type

String

Example

```
$
v[1] = Directory.GetAttributes("Test.txt")
```

SetAttributes

Description

With this method you can set the attributes of a file/directory. You can use (and combine) the file attribute constants for this. You can't convert directories into files (or vice versa).

Syntax

```
Directory.SetAttributes(Path, NewAttributes)
```

Return value

None

Type

Method

Parameter: Path

Description

Specifies the (relative) path of the file/directory whose attributes shall be set. You can only set the attributes of files/directories inside of the game directory (and subdirectories).

Data type

String

Parameter: NewAttributes

Description

Specifies the new attributes of the file/directory. You can use (and combine) the file attribute constants for this. Files may not have the attribute FILE_ATTRIBUTE_DIRECTORY.

Data type

Dword

Example

```
$
Directory.SetAttributes("Test.txt", FILE_ATTRIBUTE_HIDDEN)
```

At the end the file "Test.txt" whould be marked as hidden.

FindFirst

Description

With this method you can start to browse a directory. You can specify the search options with the parameter search pattern. To continue the browsing (and even find all files/directories) you must call the method FindNext in a loop. During each browsing you find first the directories "." and "..". "." is the current directory and ".." is the parent directory. If you don't need this information you should skip it. This method is equivalent to the windows function FindFirstFile.

Syntax

```
Directory.FindFirst(SearchPattern)
```

Return value

String

Type

Method

Parameter: SearchPattern

Description

Specifies the (relative) path of the browsing directory including the placeholders. You can use the asterisk (*) and the question mark (?) as placeholder. The asterisk stands for any number of unknown chars and the question mark stand for exact one unknown char. To browse effectively you must use at least one of these placeholders. You can only access files/directories inside of the game directory (and subdirectories).

Data type

String

Example

```
$
a[1] = Directory.FindFirst("*.lmu")
```

At the end a[1] whould contain the filename of the first map of the game (= files in the gamedirectory with the extension lmu are maps).

FindNext

Description

After starting to browse a directory using the FindFirst method you can continue browsing using this method. After each call this method returns the filename of the next matching file/directory. If this method returns an empty string then the browsing is finished and it must be closed using FindClose.

Syntax

```
Directory.FindNext()
```

Return value

String

Type

Method

Example

```
$
a[1] = Directory.FindNext()
```

At the end a[1] whould contain the next matching file/directory of a browsing. This command should be used in a loop until it returns an empty string (in this example a[1] whould be empty).

FindClose

Description

With this method you can close a previously started browsing.

Syntax

```
Directory.FindClose()
```

Return value

None

Type

Method

Example

```
$
Directory.FindClose()
```

10. Error messages

Description

During the execution of a DestinyScript the can occur errors on different places. A list of all errors (and their meanings) is listed here.

List of errors

Nummer	Name of constant	Short description
0	ERROR_UNKNOWN	An unknown error
1	ERROR_SYNTAX	Unexpected chars are used in the DestinyScript
2	ERROR_NOVALUE	A value is expected but no one is denoted
3	ERROR_UNKNOWNNAME	An unknown name was used
4	ERROR_CONVERT	A value couldn't be converted due to an unknown reason
5	ERROR_READONLY	It was tried to write a read-only value
6	ERROR_ARRAYBOUND	The used index exceeds the boundaries
7	ERROR_RANGE	A value couldn't be converted because it is out of the range of the destination data type
8	ERROR_MEMORY	The isn't sufficient free memory available
9	ERROR_VALUE	An invalid value has been used for a parameter
10	ERROR_BINARYFLOAT	It was tried to execute a binary operation with a floating point number
11	ERROR_CALCSWITCH	It was tried to calculate with a switch

12	ERROR_CALCSTRING	It was tried to calculate with a string
13	ERROR_FLOATERROR	An error occured during a floating point operation
14	ERROR_FLOATLENGTH	To much places have been denoted for a floating point number
15	ERROR_DIVISIONBYZERO	It was tried to divide by zero
16	ERROR_STRINGFORMAT	A string has an invalid format so it couldn't be converted
17	ERROR_STRINGRANGE	It was tried to access a char which exceeds the length of the string
18	ERROR_PICTURE	A picture hasn't been loaded
19	ERROR_PIXELRANGE	It was tried to access an area that exceed the boundaries of a picture
20	ERROR_SAMEPICTURE	It was tried to use the same source and destination picture
21	ERROR_PALETTERANGE	It was tried to access a palette entry which exceeds the 256 colors palette
22	ERROR_SOCKETSTARTUP	The socket system couldn't be initialized
23	ERROR_NOFREESOCKET	There are no more free sockets available
24	ERROR_CANTCREATESOCKET	It was not possible to create a socket

25	ERROR_SOCKETSTILLOPEN	It was tried to open a socket while its already open
26	ERROR_SOCKETNOTOPEN	It was tried to use a socket method which requires an open socket
27	ERROR_CANTCONNECT	It was not possible to connect to the specified address
28	ERROR_SOCKETTYPE	It was tried to use a socket method which is appointed for an other socket type
29	ERROR_SOCKETERROR	An unknown error occured during the access of a socket
30	ERROR_OOB	Out of band data (garbage) has been received on a DestinySocket
31	ERROR_STRINGTOOLONG	It was tried to send a string which is longer than 255 bytes
32	ERROR_NOFREEFILEHANDLE	It was tried to open an already open file handle
33	ERROR_CANTRESOLVEPATH	It was not possible to resolve a path
34	ERROR_NOPERMISSION	It was tried to access a path outside of the game directory
35	ERROR_CANTOPENFILE	It was not possible to open a file
36	ERROR_FILENOTOPEN	It was tried to use a file function which requires

		an open file
37	ERROR_CANTACCESSFILE	It was not possible to access a file
38	ERROR_CANTCREATEDIR	It was not possible to create a directory
39	ERROR_CANTREMOVEDIR	It was not possible to remove a directory
40	ERROR_CANTRENAMEFILE	It was not possible to rename/move a file/directory
41	ERROR_CANTCOPYFILE	It was not possible to copy a file
42	ERROR_CANTDELETEFILE	It was not possible to delete a file
43	ERROR_CANTREADATTRIBUTES	It was not possible to retrieve the attributes of a file/directory
44	ERROR_CANTWRITEATTRIBUTES	It was not possible to set the attributes of a file/directory
45	ERROR_SEARCHSTILLOPEN	A directory browsing is already started
46	ERROR_CANTSTARTSEARCH	It was not possible to start a directory browsing
47	ERROR_NOSEARCHSTARTED	It was tried to use a method which requires an already started directory browsing

Error 0: ERROR_UNKNOWN

Description

This error may never occur. It is only here to cover the impossible case.

Error 1: ERROR_SYNTAX

Description

This error occurs if an invalid char has been used in DestinyScript. This could happen whether there are too much parameters declared (in that case the interpreter want's a closing parenthesis and not a comma).

Example of the error

```
$
:
```

Error 2: ERROR_NOVALUE

Description

This error occurs if a value is required but no one is specified. This could be an empty pair of parantheses in a formula or simply a missing term.

Example of the error

```
$
v[1] = 3 + () + 1
```

Error 3: ERROR_UNKNOWNNAME

Description

This error occurs if a name (this means name of an object, method, property, constant or scope) has been denoted that doesn't exist.

Example of the error

```
$
v[1] = Picture[1].Toast
```

The Picture object has no property with the name "Toast".

Error 4: ERROR_CONVERT

Description

This error may never occur. It whould only occur if the interpreter tries to convert a not specified value.

Error 5: ERROR_READONLY

Description

This error occurs if it is tried to write a read-only (this means write-protected) value. Wheter a value is read-only or not is written in its definition.

Example of the error

```
$
Destiny.DllVersionMajor = 100
```

The property "DllVersionMajor" of the Destiny object is declared as read-only.

Error 6: ERROR_ARRAYBOUND

Description

This error occurs if an invalid index is used. For example if the range is defined as 1 to 100 then all indices less than 1 or bigger than 100 are invalid.

Example of the error

```
$
Picture[-1].UseMaskColor = False
```

The index of the Picture object start with 1. Hence each negative index is invalid.

Error 7: ERROR_RANGE

Description

A data type with a huge value was tried to be converted into a data type with a small range. For example the data type byte allows only values in the range from 0 to 255. So it is not possible to convert a value smaller than 0 or bigger than 255 into a byte.

Example of the error

```
1 $
2 f[1] = Math.Exp(11);
3 v[1] = f[1]
```

The error occurs in line 3. f[1] is 100,000,000,000 but the maximum value of dword is 2,147,483,647. Hence it is not possible to store the huge value into the small data type.

Error 8: ERROR_MEMORY

Description

There is not enough memory available to execute a command. For example this could happen if it is tried to allocate a string which is some giga bytes long. This error can depend on the used target computer system where the game is running. In this case a reboot of the computer could help.

Example of the error

```
$
a[1] += a[1] + String.Fill("This is just" +
" a long example string", 1000)
```

If this DestinyScript is executed in a (endless) loop then a[1] will grow and grow and grow... In theory this could raise the error if there is insufficient memory.

Error 9: ERROR_VALUE

Description

An invalid value has been specified for a parameter. Which values are valid for a parameter is written in the definition of its method. For the most parameters, where ranges are definied, you can use constants.

Example of the error

```
$
Server.Listen(1000000, SOCK_DESTINY)
```

The first parameter of the listen methods specifies the port where the socket will listen on. The range of this port specified as 1 to 65535. The value 1000000 exceeds this range.

Error 10: ERROR_BINARYFLOAT

Description

This error occurs when it is tried to apply a binary operation (AND, OR, NOT and XOR) with a floating point number. Binary operations are only allowed for integer data types (byte, word and dword).

Example of the error

```
$
f[1] &= 1
```

The used operator is an AND operator and not valid for an operation with the floating point number f[1].

Error 11: ERROR_CALCSWITCH

Description

This error occurs if it is tried to calculate with switches. This includes arithmetical and binary operations.

Example of the error

```
$
s[1] += 1
```

The used operator is an addition operator. Because s[1] is a switch this is invalid.

> ***Information***
> *To apply logical operations with switches you can use the Logic object.*

Error 12: ERROR_CALCSTRING

Description

This error occurs if it is tried to calculate with strings. This includes arithmetical and binary operations. The only operator, which may be used with strings (additionally to the set operator), is the addition operator which is used to concatenate strings.

Example of the error

```
$
a[1] = "Hello Nr. " + 5
```

Numbers and strings are used in the same calculation. So the addition operator is interpreted as addition and not as concatenation.

> ***Information***
> *To avoid this error you can convert strings into numbers first (or vice versa). You can use the Convert object for this.*

Error 13: ERROR_FLOATERROR

Description

This error occurs if a floating point operation was invalid. This could happen with (invalid) floating point numbers which are read from a file.

Error 14: ERROR_FLOATLENGTH

Description

This error occurs if a number (in text form), which is longer than 13 chars, is being converted into a floating point number.

Example of the error

```
$
f[1] = 1234567890.1234567890
```

The number has 10 integer places and 10 decimal places. In sum this are 20 places. This are more than the maximum allowed 13 places.

Error 15: ERROR_DIVISIONBYZERO

Description

This error occurs if it is tried to divide trough zero. This is (depending on the laws of mathematics) not possible. In theory it is possible to subtract zero infinite times from any number.

Example of the error

```
$
v[1] /= 0
```

Error 16: ERROR_STRINGFORMAT

Description

This error occurs if it is tried to convert a string into a number which has an invalid format (this means it doesn't contain (only) a number).

Example of the error

```
$
v[1] = "Number: 12345"
```

The string "Number: 12345" is not a number. Instead "12345" whould be valid.

Error 17: ERROR_STRINGRANGE

Description

This error occurs if it is tried to access a position which exceeds the length of a string.

Example of the error

```
$
v[1] = String.Ord("Hello", 5)
```

The string "Hello" has a length of 5 chars. It is tried to access the sixth char (= position 5) with the Ord method.

Error 18: ERROR_PICTURE

Description

This error occurs if it is tried to access a picture property, which is only available if the picture has been loaded (e. g. the pixels of a picture).

Example of the error

```
$
v[1] = Picture[1].Pixel[0, 0]
```

If there is no picture loaded as id 1 this will raise an error.

Error 19: ERROR_PIXELRANGE

Description

This error occurs if it is tried to access some pixel which exceed the picture's boundaries.

Example of the error

```
$
v[1] = Picture[1].FillRect(0, 0, 100, 100)
```

The error whould occur if the Picture no. 1 whould be smaller than 100 x 100 pixels (e. g. if the picture has a size of 20 x 20 pixels).

Error 20: ERROR_SAMEPICTURE

Description

This error occurs if it is tried to copy an area from one picture to the same picture.

Example of the error

```
$
Picture[1].CopyRect(0, 0, 1, 0, 0, 100, 100)
```

The error whould occur because the source picture (= 1) is the same as the destination picture (= 1).

Information
To avoid this problem you could load the same picture two times. Then you could copy the area from one picture to the other and then erase the copy.

Error 21: ERROR_PALETTERANGE

Description

This error occurs if it is tried to access a palette entry outside the range of 0 to 255.

Example of the error

```
$
Picture[1].Palette[256] = 0xFF
```

The palette entry with the index 256 doesn't exist.

Error 22: ERROR_SOCKETSTARTUP

Description

This error whould occur if it is not possible to initialize the socket system. This is an error of the target system, where the game is running. The reason for this error could be a wrong winsock version or an invalid network driver.

Error 23: ERROR_NOFREESOCKET

Description

This error occurs if there is no more free socket to accept an incoming connection.

Error 24: ERROR_CANTCREATESOCKET

Description

This error occurs if it is not possible to create a socket. This is an error of the target system, where the game is running. The reason for this error could be insufficient resources.

Error 25: ERROR_SOCKETSTILLOPEN

Description

This error occurs if it is tried to connect an already connected socket.

Example of the error

```
$
Client[0].Connect("127.0.0.1",
SOCK_DESTINY);
Client[0].Connect("127.0.0.1", SOCK_DESTINY)
```

If we assume that the first connection whould be established then the second try to establish a connection whould raise this error. To avoid this error it is satisfactory to close the socket with the Close method before the connection is being established.

Error 26: ERROR_SOCKETNOTOPEN

Description

This error occurs if it is tried to do an operation which requires a connected socket.

Example of the error

```
1 $
2 Client[0].Close();
3 Client[0].SendVariable(1, 1)
```

The error whould occur in line 3, because the socket is closed (line 2). Hence it is not possible to send dara.

Error 27: ERROR_CANTCONNECT

Description

This error occurs if it was not possible to establish a connection.

Example of the error

```
$
Client[0].Connect("255.255.255.255",
SOCK_DESTINY)
```

It is not possible to connect to the specified address (in this case it is an invalid address).

Error 28: ERROR_SOCKETTYPE

Description

This error occurs if it is tried to use RAW methods on a DestinySocket or DestinySocket methods on a RAW socket.

Example of the error

```
1 $
2 Client[0].Connect("127.0.0.1",
3 SOCK_DESTINY);
  Client[0].SendRawData("Hello", 5)
```

The error whould occur in line 3 because the socket type is DestinySocket and the SendRawData method is only for RAW sockets.

Error 29: ERROR_SOCKETERROR

Description

This error occurs if the socket system reports an error. This is a problem of the target system, where the game is running. The reason could be insufficient resources.

Error 30: ERROR_OOB

Description

This error occurs if a DestinySocket receives invalid data (out of band). This could occur if a DestinySocket connects to a RAW socket (or vice versa).

Information *The DestinyProtocol has no error handling. Hence a connection must be closed an re-established if such an error occurs. Usually the used TCP/IP protocol avoids the arrival of invalid data packages (because the TCP/IP protocol has its own error handling).*

Error 31: ERROR_STRINGTOOLONG

Description

This error occurs if a string with a length greater than 255 bytes shall be sent over a DestinySocket.

Example of the error

```
$
Client[0].SendString(1, String.Fill("Bla", 500))
```

The error whould occur because the string, which shall be sent, is greater than 255 bytes (the length is 1500 bytes total).

> ***Information***
> *To avoid this error you could split the string into 255 char pieces (e. g. with the SubStr method). Next you send the first piece with the string id. Finally you send the other pieces with id 0. The receiving socket could append all received strings with id 0 to the last string id that was received.*

Error 32: ERROR_NOFREEFILEHANDLE

Description

This error occurs if it is tried to open a file although the used file handle is already open.

Example of the error

```
1 $
2 File[0].Open("Test.txt", FILE_WRITE);
3 File[0].Open("Test2.txt", FILE_WRITE)
```

The error whould occur in line 3, because the used file handle (= 0) is already open (line 2).

Error 33: ERROR_CANTRESOLVEPATH

Description

This error occurs if it was not possible to resolve the path. This is an error of the target system, but could occur in theory with invalid paths.

Error 34: ERROR_NOPERMISSION

Description

This error occurs if it was tried to access a file or folder, which is outside of the game directory.

Example of the error

```
$
Directory.DeleteFile("C:\NTLDR")
```

Unless the game is running in C:\ (and this whould be stupid) this whould raise an error. Otherwise (if this whould be possible) an execution of this command could waste the computer system after a reboot. This is the reason why file/directory access is only allowed inside of the game directory.

Error 35: ERROR_CANTOPENFILE

Description

This error occurs if an file couldn't be opened. The reason could be that the file is already opened by an other program, because an not existing file is being opened only with read access, because the file name includes invalid chars, ...

Example of the error

```
$
File[0].Open(
"Filenames may not contain ?.txt", FILE_READ)
```

The error occurs because filenames may not contain question marks.

Error 36: ERROR_FILENOTOPEN

Description

This error occurs if a method is called which requires an open file handle, but the used file handle is not open.

Example of the error

```
1 $
2 File[0].Close();
3 File[0].WriteRawData(12345, TYPE_DWORD)
```

The error whould occur in line 3, because the file handle is closed (line 2).

Error 37: ERROR_CANTACCESSFILE

Description

This error occurs if a read/write command on a file handle fails. This depends on the target system, where the game is running. The reason could be insufficient free disk space.

Error 38: ERROR_CANTCREATEDIR

Description

This error occurs if it was not possible to create a directory. This depends on the target system, where the game is running. The reason could be insufficient free disk space or an other file/directory with the same name already exists.

Error 39: ERROR_CANTREMOVEDIR

Description

This error occurs if it was not possible to remove a directory. Additionally to reasons of ERROR_CANTCREATEDIR the reason could be that the directory, which shall be removed, isn't empty (so it contains files/directories).

Error 40: ERROR_CANTRENAMEFILE

Description

This error occurs if it was not possible to rename/move a file/directory. This depends on the target system, where the game is running. The reason could be insufficient free disk space or that already a file/directory exists with the target name.

Error 41: ERROR_CANTCOPYFILE

Description

This error occurs if it was not possible to copy a file. This depends on the target system, where the game is running. The reason could be insufficient free disk space or that already a file/directory exists with the target name.

Error 42: ERROR_CANTDELETEFILE

Description

This error occurs if it was not possible to delete a file. This depends on the target system, where the game is running. The reason could be that the file is marked as read-only or that already a file/directory exists with the target name.

Error 43: ERROR_CANTREADATTRIBUTES

Description

This error occurs if it was not possible to retrieve the attributes of a file/directory. This depends on the target system, where the game is running. The reason could be that the user has not the required rights to access the file/directory.

Error 44: ERROR_CANTWRITEATTRIBUTES

Description

This error occurs if it was not possible to set the attributes of a file/directory. This depends on the target system, where the game is running. The reason could be that the user has not the required rights to access the file/directory.

Error 45: ERROR_SEARCHSTILLOPEN

Description

This error occurs if it is tried to open a new directory browsing while an other is still open.

Example of the error

```
1 $
2 Directory.FindFirst("*.*");
3 Directory.FindFirst("*.*")
```

The error whould occur in line 3, because the browsing is still open (since line 2).

Error 46: ERROR_CANTSTARTSEARCH

Description

This error occurs if it was not possible to open a directory browsing. The reason could be an invalid search pattern or an invalid path.

Example of the error

```
$
Directory.FindFirst("|\*.*")
```

The error occurs because there is no directory with the name | (in fact this is not possible, because a file/directory may not contain | in its name).

Error 47: ERROR_NOSEARCHSTARTED

Description

This error occurs if it is tried do retrieve the next browse result, but no directory browsing was started.

Example of the error

```
1 $
2 Directory.FindClose();
3 a[1] = Directory.FindNext()
```

The error whould occur in line 3, because the directory browsing is closed (line 2).

< Back 9.20 Directory object | 10. Error messages | **Forward >** 11. MessageLink

11. MessageLink

Description

If the MessageLink was embedded into the RPG_RT (for this see the manual of the DestinyPatcher) then are some new placeholders added to the message command. These placeholder can be used to display the content of Destiny.dll scopes. The formatting is similar to the formatting of the Format method of the String object. The differences are that you must add a backslash in the front of the placeholder, an index at the end of the placeholder and that you can use the a-placeholder for strings. All in all you have three additional placeholder with the MessageLink: A for strings, D for dwords and F for Doubles (the placeholders are equal to the identifers of the scopes).
MessageBox:

```
Hello, this is the first string: "\a[1]"
This is the second dword: \d[2]
and this is the third double: \f[3]
```

The formatting of the placeholders looks like the formatting of the default RPG-Maker placeholders. But each of these three placeholders has its own properties.

The string placeholder

Description

The only special property of the string placeholder is that it doesn't support line breaks. This depends on the RPG_RT which internal handles lines separated. A text, which has usually multiple lines, is shown in a single line. The chars, which indicate a line break, are displayed.

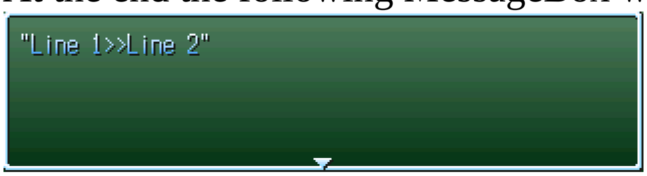
Example

```
$
a[1] = "Line 1" + CRLF + "Line 2"
```

MessageBox:

```
"\a[1]"
```

At the end the following MessageBox whould be displayed:



As you can see the two chars of a line break (CarriageReturn and LineFeet) are displayed as arrows. If a string shall be displayed over multiple lines then you must split it first. You can use the Pos method and the SubStr method of the String object for this.

The dword placeholder

Description

This placeholder can display dwords with a minimum number of digits. To do this you simply write the number of digits behind the placeholder.

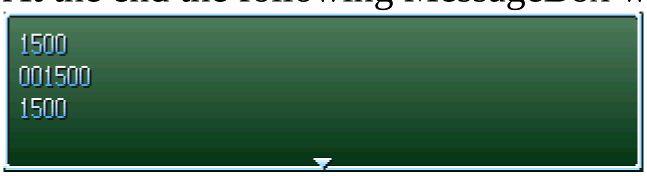
Example

```
$
d[1] = 1500
```

MessageBox:

```
\d[1]
\d6[1]
\d2[1]
```

At the end the following MessageBox whould be displayed:



In the first line the number of digits is determined automatically. In the second line we specified a minimum number of digits which is greater than the required number of digits to display the number. Hence zeros are added to the beginning of the number. In the third line the specified minimum number of digits is exceeded by the required number of digits to display the number. Hence the number is displayed normally.

The double placeholder

Description

This placeholder can display doubles with a minimum number of integer places and an exact number of decimal places. To do this you simply write the minimum number of digits for the integer places, a dot (as decimal separator) and the exact number of digits for the decimal places.

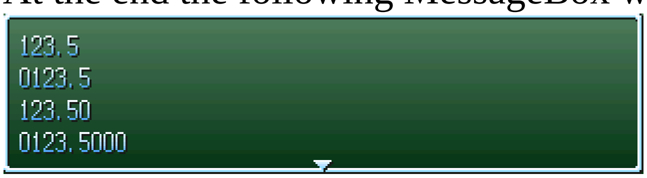
Example

```
$
f[1] = 123.456
```

MessageBox:

```
\f[1]
\f4[1]
\f0.2[1]
\f4.4[1]
```

At the end the following MessageBox whould be displayed:



In the first line the length of the integer places and the decimal places is determined automatically. In the second line a minimum length is specified for the integer places. The length of the integer places is shorter than the minimum length so zeros are added to the beginning of the number. In the third line we specified minimum 0 digits for the integer places (this is the default option) and exact 2 decimal places. In the fourth line we specified minimum 4 integer places and exact 4 decimal places.

> ***Information***
> *If a number is specified that it should not display any decimal places then only integer places will be displayed. This means that even the decimal*

separator will not be displayed. (e. g. \f0.0[1] will display 123 if f[1] is 123.456)

12. Constants

Description

At different places in DestinyScript specific numbers are required. But some values are hard to memorize. Hence you should use constants (these are names which represent the specified numbers). If you write the name of the constant instead of the number it represents then the result will be the same.

List of constant groups

Because the most constants belong together they are grouped here.

Objects	Group name	Short description
Logic	Switch values	The values 0 and 1
Math/Convert	Angle formats	Angle formats for angle methods
String	Special chars	Special chars that you can't write in DestinyScript
Error	Errors	The error constants
Keyboard	Virtual keycodes	The keys of the keyboard (and the buttons of the mouse)
Keyboard	Key state	The possible key states
Map-/Event	Special events	"Special" events (hero, boat, ...)
Map-/Event	Directions	Up, down, left and right
Picture	Actions	Actions of the pictures
Client/Server	Socket state	The possible states of a socket
Client/Server	Socket type	The possible types of a socket
Client/File	Data types	The data types of the Destiny.dll
Server	Socket choice	The next free socket
File	File mode	Modes for opening files
Directory	File attributes	Attributes of files and directories

Switch values

Description

This constants are very important, because they represent the boolean values.

Constants

Constant	Value	Description
False	0	Switch state: OFF
True	1	Switch state: ON

Angle formats

Description

This constants define the angle formats. A more specific description can be found at the Sin method of the Math object.

Constants

Constant	Value	Description
DEG	1	For angles with 360 units in a full circle (= degree)
RAD	2	For angles with π units in a full circla (= radiant)
GRAD	3	For angles with π units in a full circle (= grad)
RPG	4	For angles with 256 units in a full circle (= RPG-Maker specific)

Special chars

Description

This constants define string values, which could not be written in DestinyScript.

Constants

Constant	Value	Description
CR	ASCII char no. 13	CarriageReturn
LF	ASCII char no. 10	LineFeed
CRLF	ASCII char 13 and 10	A line break under windows
QUOTE	"	Double quotes

Errors

Description

This constants define the error numbers.

Constants

Constant	Value	Description
ERROR_UNKNOWN	0	See description of the error
ERROR_SYNTAX	1	See description of the error
ERROR_NOVALUE	2	See description of the error
ERROR_UNKNOWNNAME	3	See description of the error
ERROR_CONVERT	4	See description of the error
ERROR_READONLY	5	See description of the error
ERROR_ARRAYBOUND	6	See description of the error
ERROR_RANGE	7	See description of the error
ERROR_MEMORY	8	See description of the error
ERROR_VALUE	9	See description of the error
ERROR_BINARYFLOAT	10	See description of the error
ERROR_CALCSWITCH	11	See description of the error
ERROR_CALCSTRING	12	See description of the error
ERROR_FLOATERROR	13	See description of the error
ERROR_FLOATLENGTH	14	See description of the error
ERROR_DIVISIONBYZERO	15	See description of the error
ERROR_STRINGFORMAT	16	See description of the error
ERROR_STRINGRANGE	17	See description of the error
ERROR_PICTURE	18	See description of the error
ERROR_PIXELRANGE	19	See description of the error
ERROR_SAMEPICTURE	20	See description of the error

ERROR_PALETTERANGE	21	See description of the error
ERROR_SOCKETSTARTUP	22	See description of the error
ERROR_NOFREESOCKET	23	See description of the error
ERROR_CANTCREATESOCKET	24	See description of the error
ERROR_SOCKETSTILLOPEN	25	See description of the error
ERROR_SOCKETNOTOPEN	26	See description of the error
ERROR_CANTCONNECT	27	See description of the error
ERROR_SOCKETTYPE	28	See description of the error
ERROR_SOCKETERROR	29	See description of the error
ERROR_OOB	30	See description of the error
ERROR_STRINGTOOLONG	31	See description of the error
ERROR_NOFREEFILEHANDLE	32	See description of the error
ERROR_CANTRESOLVEPATH	33	See description of the error
ERROR_NOPERMISSION	34	See description of the error
ERROR_CANTOPENFILE	35	See description of the error
ERROR_FILENOTOPEN	36	See description of the error
ERROR_CANTACCESSFILE	37	See description of the error
ERROR_CANTCREATEDIR	38	See description of the error
ERROR_CANTREMOVEDIR	39	See description of the error
ERROR_CANTRENAMEFILE	40	See description of the error
ERROR_CANTCOPYFILE	41	See description of the error
ERROR_CANTDELETEFILE	42	See description of the error
ERROR_CANTREADATTRIBUTES	43	See description of the error
ERROR_CANTWRITEATTRIBUTES	44	See description of the error
ERROR_SEARCHSTILLOPEN	45	See description of the error
ERROR_CANTSTARTSEARCH	46	See description of the error
ERROR_NOSEARCHSTARTED	47	See description of the error

Virtual keycodes

Description

This constants define the values for keys. (Usually this whould be much more constants, but they are removed because this constants are for very special keyboard layouts)

Constants

Constant	Value	Description
VK_DOWN	40	Arrow down key
VK_LEFT	37	Arrow left key
VK_RIGHT	39	Arrow right key
VK_UP	38	Arrow up key
VK_CONTROL	17	Ctrl key
VK_MENU	18	Alt key
VK_RETURN	13	Enter key
VK_SHIFT	16	Shift key
VK_SPACE	32	Space
VK_LBUTTON	1	Left mouse button
VK_MBUTTON	4	Middle mouse button
VK_RBUTTON	2	Right mouse button
VK_NUMPAD0	96	Numeric pad 0
VK_NUMPAD1	97	Numeric pad 1
VK_NUMPAD2	98	Numeric pad 2
VK_NUMPAD3	99	Numeric pad 3
VK_NUMPAD4	100	Numeric pad 4
VK_NUMPAD5	101	Numeric pad 5
VK_NUMPAD6	102	Numeric pad 6

VK_NUMPAD7	103	Numeric pad 7
VK_NUMPAD8	104	Numeric pad 8
VK_NUMPAD9	105	Numeric pad 9
VK_MULTIPLY	106	Numeric pad multiply
VK_ADD	107	Numeric pad add
VK_SUBTRACT	109	Numeric pad subtract
VK_DECIMAL	110	Numeric pad decimal separator
VK_DIVIDE	111	Numeric pad divide
VK_0	48	0 key
VK_1	49	1 key
VK_2	50	2 key
VK_3	51	3 key
VK_4	52	4 key
VK_5	53	5 key
VK_6	54	6 key
VK_7	55	7 key
VK_8	56	8 key
VK_9	57	9 key
VK_A	65	A key
VK_B	66	B key
VK_C	67	C key
VK_D	68	D key
VK_E	69	E key
VK_F	70	F key
VK_G	71	G key
VK_H	72	H key
VK_I	73	I key
VK_J	74	J key
VK_K	75	K key
VK_L	76	L key

VK_M	77	M key
VK_N	78	N key
VK_O	79	O key
VK_P	80	P key
VK_Q	81	Q key
VK_R	82	R key
VK_S	83	S key
VK_T	84	T key
VK_U	85	U key
VK_V	86	V key
VK_W	87	W key
VK_X	88	X key
VK_Y	89	Y key
VK_Z	90	Z key
VK_BACK	8	Backspace key
VK_CAPITAL	20	Capslock key
VK_NUMLOCK	144	Numlock key
VK_SCROLL	145	Scroll lock key
VK_DELETE	46	Del key
VK_END	35	End key
VK_ESCAPE	27	Escape key
VK_HOME	36	Home key
VK_INSERT	45	Ins key
VK_PAUSE	19	Pause key
VK_PGDN	34	Page down key
VK_PGUP	33	Page up key
VK_PRINT	44	Print key
VK_TAB	9	Tab key
VK_F1	112	F1 key

VK_F2	113	F2 key
VK_F3	114	F3 key
VK_F4	115	F4 key
VK_F5	116	F5 key
VK_F6	117	F6 key
VK_F7	118	F7 key
VK_F8	119	F8 key
VK_F9	120	F9 key
VK_F10	121	F10 key
VK_F11	122	F11 key
VK_F12	123	F12 key

Key states

Description

This constants are used to specify a key state. This constants can't be used to query a key state!

Constants

Constant	Value	Description
KEYEVENTF_KEYDOWN	0	The key is pressed
KEYEVENTF_KEYUP	2	The key is released

Special events

Description

This constants are used to access special events (hero, boat, ship, airship or the current event) with the Event or Mapevent object.

Constants

Constant	Value	Description
THIS	10005	The current event
HERO	10001	The hero event
BOAT	10002	The boat event
SHIP	10003	The ship event
AIRSHIP	10004	The airship event

Directions

Description

This constants are used to identify the direction of an event.

Constants

Constant	Value	Description
DIR_UP	0	Direction: Up
DIR_RIGHT	1	Direction: Right
DIR_DOWN	2	Direction: Down
DIR_LEFT	3	Direction: Left

Actions

Description

This constants are used to identify the action of a picture.

Constants

Constant	Value	Description
ACTION_NONE	0	No action
ACTION_ROTATION	1	Rotation effect
ACTION_RIPPLE	2	Ripple effect

Socket states

Description

This constants are used to identify the state of a socket (client/server).

Constants

Constant	Value	Description
STATE_CLOSED	0	The socket is closed
STATE_CONNECTED	1	The (client) socket is connected
STATE_LISTENING	2	The (server) socket wait for incoming connections
STATE_ERROR	-1	The socket reports an error

Socket type

Description

This constants are used to identify the type of a socket (client/server).

Constants

Constant	Value	Description
SOCK_DESTINY	0	The socket is a DestinySocket (hence it uses the DestinyProtocol)
SOCK_RAW	1	The socket is a RAW socket (hence it uses not a specific protocol)

Data types

Description

This constants are used to specify the data types.

Constants

Constant	Value	Description
TYPE_VARIABLE	1	Data type: variable
TYPE_SWITCH	2	Data type: switch
TYPE_DWORD	3	Data type: dword
TYPE_DOUBLE	4	Data type: double
TYPE_STRING	5	Data type: string
TYPE_BYTE	6	Data type: byte
TYPE_WORD	7	Data type: word

Socket choice

Description

This constant is used to use the next free socket for incoming connections.

Constants

Constant	Value	Description
NEXT_FREE_SOCKET	-1	The next free socket will be used

File modes

Description

This constants are used to specify the mode for opening files. This constants can be combined using the binary OR operator.

Constants

Constant	Value	Description
FILE_READ	1	Data can be read
FILE_WRITE	2	Data can be written
FILE_APPEND	6	Data can be written and the file pointer starts at the end of the file

File attributes

Description

This constants are used to specify the attributes of a file/directory. This constants can be combined using the binary OR operator.

Constants

Constant	Value	Description
FILE_ATTRIBUTE_ARCHIVE	32	A file/directory has the archive flag
FILE_ATTRIBUTE_DIRECTORY	16	A "e;thing"e; is a directory and not a file
FILE_ATTRIBUTE_HIDDEN	2	A file/directory is hidden
FILE_ATTRIBUTE_NORMAL	128	A file/directory has no special attributes
FILE_ATTRIBUTE_READONLY	1	A file/directory can only be read
FILE_ATTRIBUTE_SYSTEM	4	A file/directory is a part of the operating system

13. Known bugs

Description

If a method of DestinyScript has not the defined effect then this is a bug (= software error). Some of these bugs depend on the target computer system, where the game is running. Hence here is a list of known bugs and solutions.

List of known bugs

Object	Methods	Source of error	Short description
Game	Save / Load	RPG_RT.exe	The game crashes if it can't read/write a save file
Picture	X / Y	RPG_RT.exe	The game crashes if the coordinates are too huge
Server	Listen	Operating system	Incoming connections are blocked
Alle mit Index	Alle	User	The game crashes on invalid indices

Game.Save / Game.Load

Source of error

RPG_RT.exe

Problem

This is a bug which can occur if it was not possible to save/load a game. In this case the game will crash. The filename of a save file is quot;SaveXX.lsd", whereas XX is the respective save slot with two digits (e. g. "Save01.lsd"). If the game crashes during save then the save file is probably read-only. If the game crashes during load then the file probably doesn't exist (or that the save file is corrupt).

Solution

To check if the save file can be read/written you can simply open it with the Open method of the File object. If the opening works then you can probably save/load the save file.

Picture.X / Picture.Y

Source of error

RPG_RT.exe

Problem

If a picture has too huge coordinates then the game will crash (e. g. X = 5000 or Y = -5000).

Solution

If a graphic shall simply not be displayed then you can place it a little bit out of the visible range of the screen. Otherwise you could split the file into small pieces and move the small pieces using DestinyScript.

Server.Listen

Source of error

Betriebssystem

Problem

This is an error which can occur with firewalls (this includes all operating systems since Windows XP, which includes a firewall). All incoming connections are blocked by the firewall.

Solution

To solve this problem you must specify that the program/port may accept incoming connections. For Windows firewalls you can use the manual on Microsoft.com. If you have an external firewall (router, etc.) you must enable the "port forwarding"e;.

All objects with index

Source of error

User

Problem

This problem occurs if an invalid index is used with objects of the RPG_RT. For example if you try to change the properties of the hero with the id 0 (the first hero id is usually 1). In such a case the game will crash.

Solution

To solve this problem you must look out to use valid indices.

14. Appendix

Description

Here shall some technical details about the Destiny.dll be descripted.
Currently this is only the design of the DestinyProtocol.

DestinyProtocol

Description

The DestinyProtocol has been developed by Bananen-Joe for the RPG-Maker. It offers the basic functions for transmitting variables, switches, dwords, doubles, words, bytes and strings. These are all associated with an id.

Assembly of a DestinyProtocol command

```
XX IIIIIIII VV...
```

The assembly is simple. At first there will be 1 byte sent as command (red). This command decides the format and the length of the value. The first parameter of the command is a dword value (little endian), which contains the associated id (blue). The second parameter of the command contains the value of the data type (purple).

List of the DestinyProtocol commands

Command (byte)	Size (of the value)	Format (value)	Description
V	4	Dword, little endian (= 4 bytes total)	A variable is sent
S	0	*none* (= 0 bytes total)	A switch is sent with value 1 (True)
s	0	*none* (= 0 bytes total)	A switch is send with value 0 (False)
D	4	Dword, little endian (= 4 bytes total)	A dword is sent
F	8	Double, little endian (= 8 bytes total)	A double is sent
W	2	Word, little endian (= 2	A word is sent

		bytes total)	
B	1	Byte (= 1 byte total)	A byte is sent
A	1 + n bytes	See string format	See string format

String format

The string format doesn't contain numbers. Hence the first byte (after the index) specifies the length of the string. The following bytes are the string. Here you can see the weakness of the format: strings, that are longer than 255 bytes, can't be sent with this format. But this is intended, because the interal buffer of the Destiny.dll has a maximum size of 500 bytes. Otherwise, if the strings could be longer than 500 bytes, it whould not be possible to check if the entire string has been received. This was the simpliest solution to ensure that not too much memory is required. Furthermore the developer of the code was lazy here.

Examples

SendVariable

```
$
Client[0].SendVariable(1, 2)
```

The SendVariable example whould create the following data package (Hex-Dump):

```
56 01 00 00 00 02 00 00 00
```

SendSwitch

```
$
Client[0].SendSwitch(100, True);
Client[0].SendSwitch(-200, False)
```

The SendSwitch example whould create the following data package (Hex-Dump):

```
53 64 00 00 00
73 38 FF FF FF
```

SendDword

```
$
Client[0].SendDword(10000, 0x12345678)
```

The SendDword example whould create the following data package (Hex-Dump):

```
44 10 27 00 00 78 56 34 12
```

SendDouble

```
$
Client[0].SendDouble(0xAABBCCDD, Math.Pi)
```

The SendDouble example whould create the following data package (Hex-Dump):

```
46 DD CC BB AA 18 2D 44 54 FB 21 09 40
```

SendWord

```
$
Client[0].SendWord(0x987654, 100)
```

The SendWord example whould create the following data package (Hex-Dump):

```
57 54 76 98 00 64 00
```

SendByte

```
$
Client[0].SendByte(0xD0C0B0A0, 3)
```

The SendByte example whould create the following data package (Hex-Dump):

```
42 A0 B0 C0 D0 03
```

SendString

```
$
Client[0].SendString(1111111, "Hello")
```

The SendString example whould create the following data package (Hex-Dump):

```
41 47 F4 10 00 05 48 65 6C 6C 6F
```

< Back 13. Known bugs | 14. Appendix | **Forward >** 15. Closing words

15. Closing words

As the great end of this manual I want to add a review about the DestinyPatch and its development. Since some years I had the idea to develop a patch for the RPG-Maker, but at that time I had not the required knowledge. I had tried to create a solution with a second exe file, which modifies the memory of the real exe file. This solution has a lot of problems (not only due to the performance). For example the memory addresses differ in the RPG_RT depending on the number of variables allocated, etc.. An other problem whould be the parallel monitoring of memory. This could make trouble on slow computer systems.

At that time I had the idea that a patch, which could read the content of the comment command, whould be optimal. This whould have the advantage that each command, which shall be executed by the patch, is entered into the RPG-Maker. And further that the patch is working serial (so the RPG-Maker whould make a pause during the execution of a patch command). Three years later (end of the year 2005 - at that time I had gathered much more knowledge about exe files and learned the programming language Assembler) I picked up my old idea and started to develop the DestinyPatch. To do this I had disassembled the RPG_RT.exe and patched it by hand (with a hex editor). The patch worked, but it was poor. For example it didn't afford a real formula interpreter. Additionally some functions (like the disabling of the F12 key) made trouble. In my opinion the patch was more an impertinance than a helpful tool. Hence I canceled the DestinyPatch.

At the beginning of 2007 I rediscovered the pleasure of modifing games. I modified a game for a friend (more life, etc.). Hence I resumed the project "Destiny" and completely revised. But this time I used more structure! For example I created some structograms for the formula interpreter. This eases the work enormously. Additionally I solved the problems like the F12 key thingy on the first start-up. Sometimes it is useful to make a (short) break. The entire year 2007 I worked continously (this means at leas 1 hour per week) on the patch until it was ready at the end of 2007.

The most difficult part of the project was the development of the help files. (Writing easily readable text is much more complicated than writing procedures which modify the stack of its own calling procedures). But the documentating of the patch included advantages. Because I noticed some meaningless functions, that could be removed, and some missing functions, that should be added (e. g. the If method). By hindsight some functions, which are from the first project time of 2005, are useless. But I leave them inside because they could be possibly useful (e. g. who needs a arcus secans?!).

Now (Beginning of 2008) the patch is checked ready (for now). I don't know if there will be new versions. (This depends on the requirement of the users). Furthermore some people whould like to have functions, which are hard to include. (For this functions it whould be easier and much more expedient to write an own RPG-Maker).

But for now I wish all people, who want to use the patch (nevertheless to the old RPG-Maker 2000 version) a lot of fun!

16. Imprint

The entire DestinyPatch (Destiny.dll and DestinyPatcher) and the help files (DestinyPatcher help file and DestinyScript help file) have been developed by David Gausmann (alias Bananen-Joe). Closing words about the project can be found in the help file for the scripting language (DestinyScript).

If you have questions (which could not be answered with these help files), remarks and critique you can contact the author via his email address: DestinyDLL ette Bananen-Joe.de (You must replace the ette with an at symbol - if you don't understand this please don't contact me!). Questions send over other ways (other email addresses, messenger, ...) won't be answered and (as far as possible) immediately deleted!

And have a look on my homepage!
http://www.bananen-joe.de/

Have fun - yours sincerely Bananen-Joe.