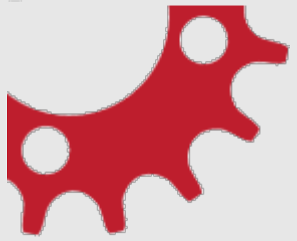
Thread: Expression Gate Documentation

1 Likes

04-18-2007 #1

Syranide
Expressionism 2.0
WIREMOD WM DEVELOPER



Join Date:	Mar 2007
Location:	Sweden
Posts:	4,566

Expression Gate

Documentation 1.06

- Added vector support
- Added section "Internal Clock"
- Added -orb-'s expression gate tutorial
- Added a short tip to conditional expressions
- Added function reference
- Added starter examples
- Added section "The Gate"
- Added section "Inputs and Outputs"
- Changed the layout for examples
- Minor change in the "Packet Support" section

[topic=188]**Official Thread for the Expression Gate**[/topic]

If you have any problems or questions at all regarding the expression gate or having problems understanding the expression language, post here and I will help you the best I can.

For beginners that feel lost in pages of documentation, here is a superb tutorial on starting with the Expression Gate, done by -orb- over at Tricky's tutorials, **Expression Gate Video Tutorial**. Also, **http://www.wiremod.com/4890-revans-wiremod-tutorials.html** by turck3 has very good progession and should allow anyone to get started.

For further inspiration have a look at the [topic=188]official thread[/topic] at what other users have created!

Here is the documentation I wrote for the expression gate as I know it was much needed, I tried to be as detailed as possible, if there is anything unclear or confusing, just ask and I will update or improve it. I know some things might not be very neatly written and so on, so just poke me! I will probably have to extend this later, but it will do for now and I hope this should clear up any problems for beginners, people familiar with any kind of programming can probably skip through half the documentation.

If I get time, I will try to whip up some examples sometime soon showing how to use it unless someone else feels overly ambitious. Simple examples are provided with the expression gate and I am happy to answer all your questions. If you have suggestions on examples to include at the end of this documentation I would be happy to do so, but remember, they need to be rather simple to setup and preferably useful or good knowledge.

Introduction

The expression gate does not really introduce any new possibilities outside those previously available, what it does however is to allow you to create much more compact and complex setups using very few gates while at the same time allowing you to easily reuse previously created expressions. So in practice it combines the possibilties of all the gates into one, although using some of the more advanced functionality you can also use it to replace latches and certain memory components.

The CPU chip:
To clear up any misunderstandings as to the distinction between the expression gate and the CPU chip, it is first important to understand what a gate is, a gate is a single component that takes a number of inputs and produces an output based on that, in theory for the same input you should always get the same output. In contrast, a chip does not necessarily produce a specific output for specific input, it has internal states and can act on its own, like the pulser and the memory. As you might know, the expression gate is not a true gate by nature because it can keep has internal variables, but in every other sense it is a gate, it cannot act on its own, whereas the CPU chip can continously change the output and calculate stuff internally without, in theory, having any inputs. So while the CPU chip might continously update the outputs, the expression gate always waits for input before it does so.

The main idea you have to recognize is that the expression gate is a calculator and the CPU chip is a computer, the CPU chip can do everything the expression gate can. The most important differences between the expression gate and the CPU chip is that CPU chip can be used for advanced states, algorithms and functionality that requires memory, loops, recursion, etc. The expression gate is linear in the sense that every line gets calculated at most once every execution and has a very limited ability to remember states.

As I'm running out of imagination, I will give some rather obvious examples here just to make it simple, it is important to note that these are only recommendations for normal contraptions and no means a rule, it is very possible that you make a hyper advanced guided missile that should be done using the CPU chip or anything that would use hi-speed links for that matter.

> Originally Posted by **Expression gate**
>
> *"Simple statistics" - simple calculations based on input, session variables allows for calculating totals, delta, etc*
> *"Guided missile" - requires no advanced states and only requires continous calculations based on given parameters*
> *"Wire scribe" - click here courtesy of Qjet*
> *For further inspiration have a look at the [topic=188]official thread[/topic] at what other users have created!*

> Originally Posted by **CPU chip**
>
> *"Radar" - although this can to some extent be done using the expression gate, the expression gate cannot use hi-speed links at the moment*
> *"Operating system" - requires advanced states, memory and independent execution*

Summary:
As I hope is clear by now, if you disregard specifics and theory, the CPU is basically superior in every way except for when it comes to programming, where the expression gate is much simpler to use with the simple constructs and mathematical equations, but has limited functionality. Remember, the expression gate does not truly add new possibilities, the CPU chip does, the expression gate is mainly for convenience and the ability to make more complex setups. I hope this is a fair comparison of the expression gate and the CPU chip, if you have objections or better examples then please notify me, that being said I think the CPU chip is a great component which allows you to create highly advanced contraptions.

If you are unsure, a simple hint is; if it can be done using gates and latches it probably should be done using the expression gate.

The Gate

The expression gate itself works like any other gate, except that you need to enter a valid program to be able to spawn it, this is done using the "New Expression..."-button in the interface which allows you to enter expressions. Once you have spawned a gate, you can upload a new program to it by simply clicking it again with the left mouse button. You can also fetch the program from the gate using the right mouse button and it will then show up in the expression gate interface. Using the reload button you can reset all session variables to zero in the expression gate (described in "Inputs and Outputs").

As long as you keep the same inputs and outputs when you upload a new program the wires will not disconnect, removing an input or an output will disconnect that wire from the expression gate.

If there is an error in your program you will be presented with a notification of what the error is, the errors are however not predictive or especially smart, they only refer to what parser expected to find, or not find. Meaning that if you get "Expected (() near (,)", it means that it was expecting a "(" before the "," in "A = fun, B = 0", in this case it is because "fun" is a function call and as such requires that you put parenthesis behind it as "A = fun(), B = 0".

Variables

Variables are ways of storing and passing values, just like the memory (M1, M2, etc) in calculators.

Caution: All variable names must begin with an Uppercase character to be correctly interpreted. Of further notice is that all variables that are not bound to input variables will be saved between each execution, which means that if you set a non-input variable to one, it will stay one until you change it, this allows you to create very complex setups as you know have limited internal memory to use.

Code:
```
A
```

Code:
```
Out
```

Code:
```
TotalDistance
```

Inputs and Outputs

Inputs and outputs are variables just like any other, however, there is one important thing about them and that is that they are bound to the inputs and outputs of the expression gate.

Meaning that for inputs, whatever is connected to input "A" on the expression gate trigger an execution of the expression every time it changes values, and the value is also assigned to variable "A" for so that it can be used during that execution. Likewise for outputs, whatever is connected to output "B" of the expression gate receives whatever value "B" was been assigned during when the last execution ended.

Both inputs and outputs are specified as a sequence of variables delimited by a single space.

Code:
```
A B C
```

If you ever look in the files of saved expressions or expresisons posted on boards, you will find a number of lines at the start with an "@", these are definitions for label (N@), inputs (I@), outputs (O@).

However, you can also assign variables that are neither inputs nor outputs, these are called session variables and are kept between each execution meaning that if you assign "C" a value, next time the expression is executed "C" will still have that value. This allows you to do some even more complex things, but it is also dangerous, because it means that you have to reset all variables at the start of each execution!

Code:
```
# BAD! Once Speed has reached 50 then Break will always be 1
Speed >= 50 -> Break = 1;
```

Code:
```
# GOOD! Break will revert to 0 once Speed is less than 50
Break = 0, Speed >= 50 -> Break = 1;
```

Code:
```
# BETTER! Same but without zeroing first
Break = (Speed >= 50 ? 1 : 0)
```

There are also constants, variables which never change their value (except by the reprogramming the gate), these are used to get rid of "magic numbers" in the code, basically, numbers that you will soon forget what they do or that are specific to your contraption, such as height or some value for tweaking. So instead of writing the actual height of the contraption where needed, you assign "HEIGHT" the value "500" (all uppercase is just a recommendation) at the start of your expression, and then use "HEIGHT" instead in your expressions.

Functions

Certain functions are provided to give you access to additional functionality, that is needed when creating certain setups, all function names start with a lowercase character followed by a left parenthesis, a comma separated argument list and a final right parenthesis "atan2(4, 5.2)".

The entire list of functions and their description is available at the end of this documentation.

Caution: Calling an undefined function will not yield an error in the current implementation, it will instead return the value "-1".

Code:
```
# Absolute value of -2, equals 2
abs(-2)
```

Code:

```
# Round the value 3.42 down, equals 3
floor(3.42)
```

Code:

```
# Convert 67 from degrees to radians, equals 1.169...
rad(67)
```

Arithmetic Operators

Arithmetic operators are the heart and soul of each expression and should not be any difficult to comprehend, they are binary operators, which mean that they take two values as input and output one, but in an understandable sentence, it just means that you can add and multiply values.

Code:

```
N + N : addition
N - N : subtraction
N * N : multiplication
N / N : division
N % N : modulo, also referred to as the remainder of division "4 / 5 = 2.5 => 0.5", note "-1 % 3 = 2"
N ^ N : exponentiation, "X to the power of 2" which is also referred to as X squared
-N    : negation, change the sign of the value
```

To those further mathematically interested all operators are left-recursive, meaning that "2 / 2 / 2" means "(2 / 2) / 2" and not "2 / (2 / 2)", if you are uncertain of how it applies then always use parenthesis.

Code:

```
# Add 2 to 2, equals 4
2 + 2
```

Code:

```
# Multiply 4 by 2, equals 8
4 * 2
```

Code:

```
# Divide -2 by -4, equals 2
-4 / -2
```

Code:

```
# Add 4 to 2 then square it, equals 36
(4 + 2) ^ 2
```

Assignment Operators

Assignment operators are used for assigning values to variables, in theory there exists only one such operator, namely the assignment operator "=".

However, assignment operations where you do not replace the current variable, but act upon its current value are also available and these are referred to as syntactic sugar (because they are just shorthand for a longer expression), and one of these is an assignment operations that adds a value to the variable "+=".

Code:
```
V = N  : assignment
V += N : assignment using addition
V -= N : assignment using subtraction
V *= N : assignment using multiplication
V /= N : assignment using division
V %= N : assignment using modulo
V ^= N : assignment using exponentiation
```

They are all short-hand for "Variable = Variable (op) Value" which translates to "Variable = Variable + Value" for "+=" and "Variable = Variable % Value" for "%=".

Code:
```
# Assign Out the value of A
Out = A
```

Code:
```
# Assign Out the value of A + 2
Out = A + 2
```

Code:
```
# Assign B the value of C + D, then assign Out the value of A + B, which also corresponds to A + C + D
Out = A + (B = C + D)
```

Comparison Operators

Comparison operators are used for outputting zeroes and ones when put simply, "is X less than 42", "is Y equal to 20", these are also used to create conditions which refers to creating a set of requirements for something to become active (one) which are combined using logical operators explained in the next section.

Code:
```
N == N : equal
N != N : not equal
N > N  : greater than
N < N  : less than
N >= N : greater or equal
N <= N : less or equal
```

Caution: The comparison operators in the expression gate are different from those of the other gates, namely that zero is false (0) and non-zero is true (1), which is different from that used in gates where negative and zero is false (0) and positive is (1). The cause for this is choice for programming semantics, although this might change in the future. Of further notice is that this gate as all others determine equality using a delta of 0.001, namely that "0 is equal to 0.00099" and so on.

Code:
```
# Is 2 equal to 2, equals 1
2 == 2
```

Code:
```
# Is 2 less than 1, equals 0
2 < 1
```

Code:
```
# Is 3 not equal to 3, equals 0
3 != 3
```

Logical Operators

Logical operators are of little use by themselves, but when used in conjunction with comparison operators they become a powerful tool for creating conditions. These are used for creating more complex setups such checking whether a value is in range. The same applies here as for the comparison operators, either they return true (1) or false (0), also they input false (0) or true (non-zero).

Code:
```
B & B : and, if all are true then true otherwise false
B | B : or, if any is true then true otherwise false
!B    : not, equal to "B == 0", negates the logical value "2 => 0", "1 => 0", "0 => 1"
```

Code:
```
# 1 and 0, equals 0
1 & 0
```

Code:
```
# 2 or 0, equals 1
2 | 0
```

Code:
```
# 3 greater than 5 or 4 less than 2, equals 0
3 > 5 | 4 < 2
```

Conditional Expressions

A conditional expression will execute one out of two expressions depending on the value of the condition, it allows you to do certain calculations at certain times and other calculations otherwise.

Code:
```
(B ? T : F) : if B is true then return T else return F
```

Caution: Note that you must put it inside parenthesis or you might very likely get an error.

Code:
```
# These two have identical functionality
Out = (A < 0 ? 1 : 0)
Out = (A < 0)
```

Code:
```
# If A is greater than 32 return 32, otherwise return A, then assign the value to Out
Out = (A > 32 ? 32 : A)
```

Code:
```
# If Select is true then return A + B, otherwise return B + C, then add 2 and assign the value to Out
"Out = 2 + (Select ? A + B : B + C)
```

Conditional Statements

Although the conditional expression is very handy, it is often useful to being able to include control over code-flow on another level, so that only certain parts of the code will be run at certain times. Or to halt execution if a precondition is not satisfied.

Code:
```
B -> E;           : if B is true then execute E
B -> E1, E2, ...; : if B is true then execute E1 then E2
B -> E1 E2 ...;   : alternative syntax for the above
```

While in a conditional statement you can also write "end", which will effectively stop the execution from going any further.

Code:
```
B -> end;         : if B is true then stop execution
B -> E, ..., end; : if B is true then execute E and stop execution
B -> E ... end;   : alternative syntax for the above
```

Caution: Be aware of the semicolon that must end each conditional statement.

Code:
```
# If Reset is true then reset Clock, always increment by 1
Reset -> Clock = 0; Clock += 1
```

Code:

```
# If Active is true and Angle is greater than 0 then set ThrustUp to 1 and ThrustDown to 0
Active & Angle > 0 -> ThrustUp = 1, ThrustDown = 0;
```

Code:
```
# If Tick is not triggered and true then stop execution, otherwise increment Count
!(~Tick & Tick) -> end; Count += 1
```

Sequencing

As you just have seen in the conditional statement, it is possible to execute multiple expressions by separating them by either space or comma, and the same is true outside conditional statements.

Code:
```
E1 E2 ...   : execute E1 then E2
E1, E2, ... : alternative syntax for the above
```

Code:
```
# Set A to 1 then B to 2
A = 1, B = 2
```

Packet Support

Although there is packet support through the use of the duplexer gate in wiremod, the support in the expression gate does not use the same technique and will not interact with that gate. The reason for this is that as you might've seen, they can cause quite a mess if you connect them to the wrong gate and crash debugging, although you can very easily create your own expression gate duplexer (correctly called a multiplexer/demultiplexer) using this method.

To send a packet (multiple values) over a single wire you allocate an output (Packet) in the sender gate and make a call "Packet = send(A, B, B + A)", you can specify any number of values to the function.

To receive a packet you specify an input (Packet) in the receiver gate and make a call to "B = recv(Packet, 2)" which means that you will fetch argument number "2" from the packet, which in this case is the value of "B", specifying an out-of-bounds index will return the value "-1".

Code:
```
# The sender (output: Packet), send 3 values (1: A, 2: B, 3: B + A)
Packet = send(A, B, B + A)
```

Code:
```
# The receiver (input: Packet), fetch value from the packet (2: B)
B = recv(Packet, 2)
```

These are normal functions and as such there is nothing that prevents you from having multiple packet ports on an expression gate, but note that you cannot send multiple packets over the same wire during the same execution, only the last one will arive.

Caution: This is experimental functionality, and should be used with care, although the functionality itself will not cause problems, certain setups are prone to certain problems regarding the functionality of the gate, you should clearly understand the "~" operator and have read the "Special constructs" section before attempting to create anything else than a multiplexer/demultiplexer-style expression gate. Packets will not remain over time and will be overwritten by other packets, it is important that you read the packet when you receive it (remember "~", read when triggered) and do not make any attempt to read it at a later time. Gates that only input one packet wire are not prone to any these problems and can be used without any side-effects, if you have multiple inputs of any kind you will have to use a trigger ("~") mechanism.

Good applications for this functionality is to simplify communication with individual systems so that wiring can be kept to a minimum with them, and can be easily connected and disconnected and rebuilt, creating a form of event based or modular system, perhaps some kind of central node that respond to messages and/or distribute messages to connected nodes. I strongly recommend against using it to output vectors and such to make something pretty or neat, it will only cause you headache.

Code:
```
# Multiplexer: (input: A B C D, output: Packet)
Packet = send(A, B, C, D)
```

Code:
```
# Demultiplexer (input: Packet, outputs: A B C D)
A = recv(Packet, 1)
B = recv(Packet, 2)
C = recv(Packet, 3)
D = recv(Packet, 4)
```

Internal Clock

Caution: This is experimental functionality and might be altered in the future if necessary.

Up until now, being able to setup complex circuits that can delay events and continually change has only been possible via the use of an external clock of some sort. Not only is it now possible to remove those external dependencies, but also to create even more complex setups as a result of the selected implementation.

Instead of specifying a frequency, you schedule when the next clock pulse should occur using "schedule(50)", which will cause the next clock pulse to occur in 50 milliseconds. To continually keep it executing as it would with an external pulser use "interval(50)" instead and put it at the top of the expression, this function has basically the same functionality as schedule, however it does not override previously scheduled clock pulses, instead it just ignores the call.

It is also possible to schedule a clock pulse to be executed in response to e.g. a key press "~Boom & Boom -> schedule(1000);" which will cause it to execute once in one second. The smallest delay is 20ms, values lower than 20 will be set to 20 and a value of 0 will abort any scheduled clock pulse. To know whether or not the current execution is the result of an internal clock pulse the function "clk()" has been made available and will return true (1) if the current execution is the result of an internal clock pulse or false (0) if not.

Code:
```
# Counting seconds since spawned
interval(1000)
clk() -> Seconds += 1;
```

Code:
```
# Delayed dynamite trigger
~Boom & Boom -> schedule(5000);
clk() -> Explode = 1;
```

Caution: You cannot schedule multiple clock pulses, the latter will overwrite the previous scheduled clock pulse.

Special Constructs

To support some even more complex setups, there are two unary operators at your disposal.

Code:
```
~V : true if that variable has been triggered this execution (mainly for input variables)
 $ : calculate the delta of the variable, the difference of the variable from its last change, this is roughly identi
```

And you of course wonder, what on earth would I need "~" for, first of, if you are not using variables that are kept between each execution, then you don't, but if you do this might help you solve some problems, because you have to remember that the expression gate is run for EACH changed input, every time it changes, meaning that if you have a clock with additional inputs, it will run faster when you change those inputs, therefore you will want to only increment the clock when the pulser has triggered, and not when the other inputs have.

Code:
```
# If Tick is triggered and true then increment Clock
~Tick & Tick -> Clock += 1;
```

Code:
```
# Calculate thrust output using a delta to avoid wobbling and normalize the angle to avoid spazzing at (180 / -180)
Thrust = angnorm(Angle +  * 2)
```

Additional Notes

Caution: If you are writing the expression outside of the game, make sure that your lines does not exceed 92 characters or the entire expressions might not be transferred to the server, causing strange errors... if the expression gate cannot be spawned but the validation works, this is most likely the cause.

By placing a # (hash) at the start of a line that line will be excluded from parsing, in other words, it will become a comment.

Code:
```
# This is a comment!
```

A newline in an expression is identical to a space, meaning that you can split an expression into many lines, in any way you wish.

Code:
```
A = 4 +
3 / 2
# Is identical to
A = 4 + 3 / 2
```

Remember that sometimes it is smart to use more expression gates instead of one for reusability and simplicity of the code, if an expression gate has two separate functions then it might be better to split it into two different.

Starter Examples

These examples are presented using the saved expression format (described in "Inputs and Outputs"), "I@A B C" means that the expression gate should have inputs "A B C".

Hello World

This is a starter example to get anyone going immediately, simply load it into your expression gate and connect anything to its input (such as a Constant or Advanced Input), and watch the outputs using the debugger.

Code:
```
N@Hello World
I@Value
O@Value Squared SquareRoot Positive
Squared = Value * Value
SquareRoot = sqrt(Value)
Positive = Value >= 0
```

Ticking clock

1. Connect a Pulser (TickTime: 1) to Tick
2. Connect a Button to Step, the value of the button determines how much the clock will increment/decrement
3. Connect a Button to Reset.
4. Connect screens to the 3 outputs to view the result, or use the debugger.

Code:
```
N@Clock
I@Tick Step Reset
O@Hours Minutes Seconds
~Tick & Tick -> Clock += 1;
~Step & Step -> Clock += Step;
~Reset & Reset -> Clock = 0;
Seconds = Clock % 60
Minutes = floor(Clock / 60) % 60
Hours = floor(Clock / 3600) % 24
```

Function Reference

Common

Code:
```
abs(n)          : Absolute value of the number, removes the sign
clamp(n, l, u) : Value is returned within the lower and upper bound, "max(l, min(u, n))"
frac(n)         : Fractional part
int(n)          : Integer part
```

Mathematical

Code:
```
e()             : Mathematical constant e (2.71828183)
exp(n)          : e to the power of n
ln(n)           : Logarithm with base e
log(n, k)       : Logarithm with base k
log2(n)         : Logarithm with base 2
log10(n)        : Logarithm with base 10

mod(n,c)        : Matematical modulo, "mod(-3, 2) = -1"
sgn(n)          : Sign of the number, "n < 0 => -1", "n == 0 => 0", "n > 0 => 1"

cbrt(n)         : Cube root
```

```
sqrt(n)        : Square root
root(n, k)     : n:th root
```

Rounding

Code:
```
ceil(n)        : Round up to the next integer
ceil(n, d)     : Round up to the next integer with decimal precision
floor(n)       : Round down to the previous integer
floor(n, d)    : Round down to the previous integer with decimal precision
round(n)       : Round to the nearest integer
round(n, d)    : Round to the nearest integer with decimal precision
```

Selection

Code:
```
max(n, ...)    : Returns the highest value
min(n, ...)    : Returns the lowest value
avg(n, ...)    : Returns the average of the values
sel(i, n, ...) : Returns the i:th value
```

Arbitrary

Code:
```
curtime()      : The current relative time

random()       : Random value between 0 and 1
random(l, u)   : Random value between lower and upper bound
```

Trigonometric

These functions default to degree based numbers, by prepending an "r" to the function names they will be radian based instead, such as "atan2r" or "angnormr".

Code:
```
acos(x)        : Arc cosine
asin(x)        : Arc sine
atan(x)        : Arc tangent
atan(y, x)     : Arc tangent of y/x
atan2(y, x)    : Alias for atan(y, x)
cos(d)         : Cosine
cosh(d)        : Hyperbolic cosine
sin(d)         : Sine
sinh(d)        : Hyperbolic sine
tan(d)         : Tangent
tanh(d)        : Hyperbolic tangent

angnorm(d)     : Normalize the angle, "-180 =< angle < 180"
            : (Ang180 % 360), will give "0 <= angle < 360"
deg(n)         : Convert radians to degrees
rad(n)         : Convert degrees to radians
pi()           : Mathematical constant pi (3.14159265)
```

Vector Support

These functions are only provided for expert users, the implementation is highly experimental!

Code:
```
vector(x, y, z)       : Create a vector for use with these functions

vecx(v)               : X component of the vector
vecy(v)               : Y component of the vector
vecz(v)               : Z component of the vector

vecpitch(v)           : Pitch for the vector
vecyaw(v)             : Yaw for the vector

veclength(v)          : Length of the vector
vecnormalize(v)       : Normalize the vector

vecadd(v1, v2)        : Add two vectors
vecsub(v1, v2)        : Subtract two vectors
vecmul(v1, v2)        : Multiply two vectors

vecsmul(v1, n)        : Multiply a vector by a scalar
vecsdiv(v1, n)        : Divide a vector by a scalar

vecdot(v1, v2)        : Dot product of two vectors
veccross(v1, v2)      : Cross product of two vectors
vecdistance(v1, v2)   : Distance between two vectors

vecrotate(v, p, y, r) : Rotate a vector using pitch, yaw and roll
```

Self-awareness

These functions are used to get information on the gate itself, such as position, velocity and direction.

Code:
```
extcolor(r, g, b) : Sets the color, 0-255
extcolor(r, g, b, a) : Sets the color, 0-255

extcolorr() : Returns the color (red component)
extcolorg() : Returns the color (green component)
extcolorb() : Returns the color (blue component)
extcolora() : Returns the color (alpha component)

extposx()   : Returns the position (x coordinate)
extposy()   : Returns the position (y coordinate)
extposz()   : Returns the position (z coordinate)
extpos()    : Returns the position (vector)

extvelx()   : Returns the velocity (x coordinate)
extvely()   : Returns the velocity (y coordinate)
extvelz()   : Returns the velocity (z coordinate)
extvel()    : Returns the velocity (vector)

extvelabsx()   : Returns the axis aligned velocity (x coordinate)
extvelabsy()   : Returns the axis aligned velocity (y coordinate)
extvelabsz()   : Returns the axis aligned velocity (z coordinate)
extvelabs()    : Returns the axis aligned velocity (vector)

extangp()   : Returns the angle (pitch angle, -90 to 90)
extangy()   : Returns the angle (yaw angle, -180 to 180)
extangr()   : Returns the angle (roll angle, 0 to 360, use angnorm() to make it -180 to 180)

extangvelp()   : Returns the angle (pitch angle, -90 to 90)
extangvely()   : Returns the angle (yaw angle, -180 to 180)
extangvelr()   : Returns the angle (roll angle, 0 to 360, use angnorm() to make it -180 to 180)

extdirfwx() : Returns the forward vector (x component)
```

Special

These functions are for the special functionality described in previous sections.

Code:

```
concommand('say $') : Makes the owner say the value of the variable Var

first()        : Returns true if the gate is reset

send(n, ...)   : Create a packet
recv(id, i)    : Extract a value from a packet

clk()          : Clock pulse occured
interval(ms)   : Helper function for schedule
schedule(ms)   : Schedule next clock pulse
```

Last edited by Bull; 12-16-2008 at 04:25 PM.

Share

Expression 2 (documentation), SmartSnap 1.0.0
Scenic roller coaster by Bull and Syranide

04-19-2007 #2

Yerocrg

Wire Amateur

Yerocrg's Avatar

Join Date:	Mar 2007
Location:	New Jersey, U.S.A.
Posts:	89

Thanks. This is really helpful. I was having a bit of problems with conditionals until I read this.

Share

<div class='quotetop'>QUOTE(Infinity) <{POST_SNAPBACK}></div>

Wire igniter? I think it&#39;ll turn to hell...

<div class='quotetop'>QUOTE(TAD2020) <{POST_SNAPBACK}></div>

Would only require a heavenly modified wire stool to do this.

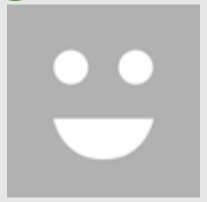
04-20-2007 #3

The support for multiple datatypes was too brief. I don&#39;t get a good feeling of the syntax because theres no examples with a description of what that example does.

Share

TomatoSoup

Wire Amateur

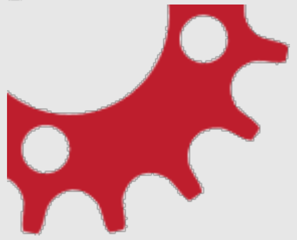


Join Date: Mar 2007
Posts: 70

04-20-2007 #4

Syranide

Expressionism 2.0



Join Date: Mar 2007
Location: Sweden
Posts: 4,566

> *Thanks. This is really helpful. I was having a bit of problems with conditionals until I read this. [/b]*

Glad to hear it!

> *The support for multiple datatypes was too brief. I don&#39;t get a good feeling of the syntax because theres no examples with a description of what that example does.[/b]*

I&#39;ve added a section for "packet support" (sending multiple values over one wire), that should be enough to clarify how to use it and for what to use it, if you still find it problematic let me know.

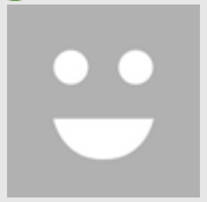
Share

Expression 2 (documentation), SmartSnap 1.0.0
Scenic roller coaster by Bull and Syranide

04-23-2007 #5

TomatoSoup

Wire Amateur



Join Date: Mar 2007
Posts: 70

Alright! Thanks!

Just one question, is this a valid line?

Input1 == 1 -> Input2 == 1 -> Out = 1;

If it works the way I want it to, then it should first check if Input1 equals 1. Then it should check if Input2 equals 1. If both those conditions meet, then it will output 1.

I just saw the ~ command, so I want to see if I can make a expression that will operate at a speed I choose by using something like ~Frequency == 1 -> as the first line. If so, how would I do that? Put a semi colon after every line? Or what?

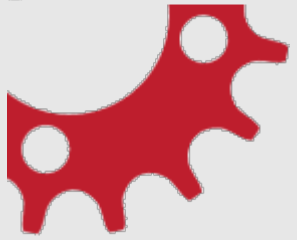
Share

04-24-2007 #6

Syranide

Expressionism 2.0

WIREMOD WM DEVELOPER



Join Date:	Mar 2007
Location:	Sweden
Posts:	4,566

> *Alright! Thanks!*
>
> *Just one question, is this a valid line?*
>
> *Input1 == 1 -> Input2 == 1 -> Out = 1;*
>
> *If it works the way I want it to, then it should first check if Input1 equals 1. Then it should check if Input2 equals 1. If both those conditions meet, then it will output 1.[/b]*

Every conditional statement (->) is terminaed by a semicolon (;), that would make your first question be

Code:
```
Input1 == 1 -> Input2 == 1 -> Out = 1;;
```

Additionally, since you what you are doing is if Input1 AND Input2 then... then you can write it as

Code:
```
Input1 == 1 & Input2 == 1 -> Out = 1;
```

Note, and a an AND (&) instead of another conditional statement, also one semicolon, because it uses a larger logical expression for the conditional statement (->) instead of two conditional statements.

Further, if Input1 and Input2 can only be true and false (button 0/1), you can write it as

Code:
```
Input1 & Input2 -> Out = 1;
```

So, Input1 == 1 in that can be translated to (Input1 != 0) ... that is it is not zero, which translates to just Input1 (note they are not equal, but for the 0/1 case they are)

> *I just saw the ~ command, so I want to see if I can make a expression that will operate at a speed I choose by using something like ~Frequency == 1 -> as the first line. If so, how would I do that? Put a semi colon after every line? Or what?[/b]*

I&#39;m not sure what you are trying to do, if you explain it further I will help you, but for now:
What the trigger-operator (~) does is that if you connect say two buttons to the expression gate, and only want to know the moment they are pressed/released, that is switch from 0 to 1 or 1 to 0 you can use the trigger-operator (~), because it tells you if that input has changed. So ~Button1 would give you something return something like the following sequence "(up) 0 0 0 0 0 (down) 1 0 0 0 0 (up) 1 0 0 0 0 (up)".

Hope this helps!

EDIT: If there was any section in my documentation that you felt was unclear please tell me so, if you have any pointers on what you had problems understanding from that secton it would be very helpful feedback to me so I can improve it. (I&#39;ve been programming seriously for many years now and have little "understanding" of what is hard to grasp for beginners)

Share

Expression 2 (documentation), SmartSnap 1.0.0
Scenic roller coaster by Bull and Syranide

04-24-2007 #7

handcrafted

Wirererer

handcrafted's Avatar

Join Date:	Mar 2007
Location:	Denmark
Posts:	211

Syranide, do you mind if i maybe use some of yours examples in the wiki pages? (the wiki page is not created yet)

Share

The Wire Wiki, please try there before asking.

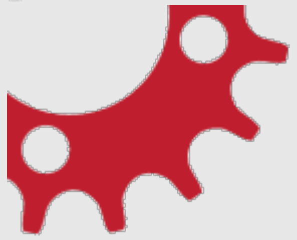
Sorry for asking but where can I edit my signature?

04-24-2007 #8

Syranide

Expressionism 2.0

WIREMOD WM DEVELOPER



Join Date:	Mar 2007
Location:	Sweden
Posts:	4,566

> *Syranide, do you mind if i maybe use some of yours examples in the wiki pages? (the wiki page is not created yet)[/b]*

If you are referring to using examples/information/whatever for the expression gate then by all means feel free to use examples and information from this documentation on the wiki. As usual, if you have any questions at all feel free to ask me.

Share

Expression 2 (documentation), SmartSnap 1.0.0
Scenic roller coaster by Bull and Syranide

04-24-2007 #9

handcrafted

Wirererer

handcrafted's Avatar

Join Date:	Mar 2007
Location:	Denmark
Posts:	211

Everybody look at the Wiki page. http://www.garrysmod.com/wiki/?title...xpression_Gate
I&#39;m working on a wiki page for the expression gate. As you can see I am writing a guide to the gate. Please help improving!

Share

The Wire Wiki, please try there before asking.

Sorry for asking but where can I edit my signature?

04-24-2007 #10

TomatoSoup

Wire Amateur

Join Date:	Mar 2007
Posts:	70

Basically, I would hook it up to a pulser to simulate a frequency variable. It would check if the frequency was true, so if I had a ticktime of .001, it would have a frequency of 1000.

That&#39;s the reason I wanted to see if Input1 == 1 -> Input2 == 1 -> Out = 1; would work. So I can make something like

~Frequency ->
rest of the Expression code

To simulate the Frequency variable on the CPU.

Nevermind...

Share

Ott likes this.

Page 1 of 40 1 2 3 11 ... Last

« Previous Thread | Next Thread »

LinkBacks (?)

Wiremod Oppaat 02-10-2010, 12:28 PM
Refback This thread

Wire Expression2:Differences to Expression1 - GMod Wiki 02-09-2010, 12:01 AM
Refback This thread

Headcrab.pl :: Zobacz temat - Expression - Szybka Nauka 02-06-2010, 09:58 AM
Refback This thread

I Want to be a Wiremod Expression Guy. - Facepunch 02-02-2010, 02:59 PM
Refback This thread

Wired Expression Chip - GMod Wiki 01-27-2010, 02:56 PM
Refback This thread

Similar Threads

!!!OLD!!! Documentation of hi-speed devices — Replies: 98
By Black Phoenix in forum CPU, GPU, and Hi-speed Discussion & Help — Last Post: 02-10-2011, 07:41 PM

!!!OLD!!! ZGPU Documentation — Replies: 38
By Black Phoenix in forum CPU, GPU, and Hi-speed Discussion & Help — Last Post: 11-29-2010, 04:54 PM

!!!OLD!!! ZCPU Documentation — Replies: 144
By Black Phoenix in forum CPU, GPU, and Hi-speed Discussion & Help — Last Post: 09-05-2010, 02:46 AM

Moongate Documentation — Replies: 24

By BlackNecro in forum Wiremod Addons & Coding Last Post: 04-22-2009, 12:52 AM

Bookmarks

- Digg
- del.icio.us
- StumbleUpon
- Google
- Facebook

Posting Permissions

You may not post new threads
You may not post replies
You may not post attachments
You may not edit your posts

BB code is On
Smilies are On
[IMG] code is On
[VIDEO] code is On
HTML code is Off
Trackbacks are On
Pingbacks are On
Refbacks are On

Forum Rules

-- Wiremod Reborn

Contact Us Wiremod.com - Home of The Wiremod Addon Archive Privacy Statement Top

All times are GMT -7. The time now is 06:50 PM.