Revan's Expression Gate 2 Guide - The Basics

Things to Notice as you Read the Guide:

There are several symbols/colors I may use throughout this guide to help you find things, so here's a list of them all:

Term - A definition you should know

Other things I may make reference to:

The Expression Gate 2 Wiki - Every now and then I will link to a page on the wiki for you to reference. It's a good idea to bookmark this page, because even experienced E2 programmers such as myself will pull up the wiki mid game to find a function. To make opening it in game easier, I recommend that you either use the steam in-game browser or run Garrysmod in windowed mode (not full screen), a setting which can be altered from options.

Introduction

Welcome to Revan's Guide to the E2 (Expression Gate 2). In reading this, it is my hope that you will better understand how and why the E2 works the way it does. I will make an attempt to keep my data here as accurate and current as possible, but considering that I no longer really play Garrysmod, I cannot guarantee that everything you see on this page will still function in the future, nor can I claim that the ways I show you how to do certain things here are the best ways to do them. The reasons for that include features being added later, my being unaware of the existence of a feature, or even just to teach you a concept, I may use a less appropriate way of achieving it for the sake of the example. Understand and respect those faults as you read.

Before you begin going through what I'm guessing will eventually amount to many pages of documentation, I advise you go through this checklist of things to have done first:

1. Have a working installation of Garrysmod
2. Have an updated copy of wiremod (This means you followed the instructions here to get it and did **NOT** try and download wiremod from garrysmod.org)
3. Be familiar with the following sections of wire tools - Familiarity means that you can use most of the items in these sections from memory or you're capable of learning from a tutorial if you need to (tutorials can be found on most things here)
    1. Physics
    2. Input/Output
    3. Detection
    4. And the various gates (memory, logic, comparison, arithmetic)
4. You've spent at least a month working with wiremod and feel comfortable with it.

Not required but may be helpful:

1. A background in programming
2. A logical mindset
3. A mathematical mindset

Now a question you probably already know the answer to, but nevertheless requires further explaining, is "What is the E2"?

The Expression Gate 2 is a programming environment that allows you to safely and easily explore the various options of Garrysmod. It allows you to use the features of almost every other wiremod component inside of one gate. That means that you don't need to create 10 logic gates for your contraption, you only need to have one Expression Gate. As you learn more about the E2, you'll be amazed further as you find out how it can even replace thrusters, rangers, and other components that were previously limited to their base components. Lastly, the Expression Gate 2, is a way to think. It requires you to perceive Garrysmod in an entirely new way, one which opens many new doors for you to explore. The fault of this however, is that many people are unable to change their perceptions, and thus are unable to use the E2. While I do not list it as a requirement, I'd like to state here, early in my guide, that if you're of a young age, you are likely to have difficulty perceiving some of the information presented below.

The most important phase I can point out in the above is "programming environment". The Expression Gate is a very limited form of programming. Now, that may scare some people, but it's actually quite easy to program, so do not fret. If you have any grasp on logic at all, programming should come easy to you.

Speaking of the environment, it's time you take a look at it.

Environment

Start Garrysmod, go to your Wiremod tab, and click on the tool for the E2. You'll see in it's options pane you have several interesting things:

- A list of all E2s you have made
    - A load and save button for each

- A New button

Click on the New Button and the programming environment will appear. There are a few things to take note of:

- The list of files you've saved on the left
- The save/exit button on the top right (may also read update/exit if you're modifying an existing E2)
- Several important lines on your E2 that we'll learn how to use later
    - @name - the name of the E2 as it appears when a player mouses over the gate - this has **no** bearing on the functionality of the gate
    - @inputs - Every piece of data that your E2 needs from other components
    - @outputs - Every piece of data that your E2 gives out to other components
    - @persists - Every variable (term will be defined later) that you need to keep between cycles (will be defined later) of your E2
    - @trigger - Relevant to cycles (term will be defined later) - don't worry about this line for now
- A large, highlighted and fairly useless comment that you should always remove.

You should always keep the four lines that start with @, but anything that comes after that is purely optional. At this point, know that if you press the save button, you will be prompted for a file name and the E2 will be saved on your computer in your '.../garrysmod/data/expression gate 2/' directory (directory name may not be exactly the same). You can also make/edit files in this directory and they will change in game (use the update button if you make new files). You do not need to transfer saves to the server to spawn their E2s like you need to with the advanced duplicator. Every line of code you write will be written below the lines that start with @.

There are several editor shortcuts which can be useful to know, you can read up on them here. One thing to know is that you cannot copy paste things to/from your E2 to/from another program on your computer. IE: You can't copy something from wiremod.com and then paste it into your E2. If you copy something in your E2, you can only paste it into the same or another E2.

Leaving the E2 editor (environment) can be done in two ways, you can

1. Click save/exit
2. Click the X in the top right corner

Personally, I prefer the second option because most E2s I write I will never use again and thus I don't want to save. As a beginner, you may want to save them, it's your choice.

After you leave the editor, if you have any code (or even if you don't), you can create your E2. To do so, click on any prop. If you entered in a name after @name, you'll see it when you look at the gate, otherwise, it'll show some basic information about the E2. If you want to edit your E2, it's fairly easy to do so, just point at it and right click; this will bring up the editor again. When you close out of the editor this time, unless you want to discard all changes you made, hit the 'update/exit' button, and do **not** press the X.

Variables (Numerical)

Variables are pieces of data that well... vary. For now, we will simplify them and just call them numbers (there are other types that we won't get into for now such as strings) that you don't know the values of when your program starts. Mathematically, if I have the statement:
answer = 4*X
Where 'X' is a variable, '*' means multiply, and 'answer' is just representing the solution and should not be confused with having any code significance, then if we state that X is 5, then answer must be 20. It's that simple. Most algebra you were taught in school will not be useful for you in 90% of all variable usages.

Creating a Variable

Creating a variable can be done in many different ways, but the easiest of which is to just set a variable equal to something, 'X=5', if entered on a line of your E2 then you will create a variable that is called 'X' and set it equal to '5'. Remember when you're setting variables that 'X=5' is NOT the same as '5=X'. '5=X' is NOT valid, you cannot set '5' equal to 'X'. '5' is a static value that cannot change and should be treated as such.
Variable creation is a simple thing, but when you do it you must remember a few things about their names:

- They must start with a capital letter or underscore (_)
    - 'Yarr' is acceptable
    - '_arr' is acceptable
    - 'ALLCAPS' is acceptable
    - 'yarr' is not
    - '3years' is not
- They can include numbers and underscores (_)
    - L33t is acceptable
- Keep them long enough to describe what they do, but short enough to not take up too much space on your lines or too long to type.
    - If you have a variable that stores the year of birth of someone, the name 'Yearofbirth' is bad because it's obnoxiously long, but 'YOB' is just as annoying to those who may read your code because they'll have no idea what that means. Instead, compromise between the two extremes, 'Year', assuming it's clear in context that it's the year of birth is acceptable, as is 'Birth', if you only store the year.

Accessing a Variable's Data

At any point in your code if you need to know the value of a variable, just type it. If you want to set the variable 'Y' equal to the value of 'X' multiplied by '2', just type 'Y=2*X' on a line.

Arithmetic Operations

You can perform many operations on variables, we've already looked at one of them, multiplication (*), but there are others which are also important. For the following examples, understand that in every instance where a constant is used (such as '5') you could replace that with the name of a variable. Also understand that almost every operator can be used with other ones, for instance, you don't need to go:

```
X=5*6
Y=X+5
```

Going 'Y = 5*6+5' is just as good.

E2 Syntax	**Mathematical Equivalent**	**Gate Equivalent**	**Example - Answers, if applicable in parentheses**
+	+	Addition Chip	5+4 (9)
-	-	Subtraction Chip	5-4 (1)
*	x (multiply)	Multiplication Chip	5*4 (20)
/	/ (divide)	Division Chip	5/4 (1.25)
^	to the power	?	2^2 (4)
%	Modulus/Remainder	Modulus	5%4 (1)
$	Delta	Delta	Delta is used with inputs to find the change in their values between cycles (see definition later). $X Is the delta of X.
~	Changed? (Delta)		This is used with inputs to see if they changed. It will be '1' if the input changed, '0' if it did not. I advise using delta instead of this in most cases. ~X

+=	N/A	Accumulator	X = 3
			X += 5 (8)
			Takes the value of the left and adding to it the value of the right and storing it in the value of the left. Thus at the end of this example, X = 8

-=	N/A	Accumulator	X=3
			X-=5 (-2)
			Remember, -2 becomes the new value of X

*=	N/A	N/A	X=3 X*=5 (15) Remember, 15 becomes the new value of X
/=	N/A	N/A	X=6 X/=3 (2) Remember, 2 becomes the new value of X

^=	N/A	N/A	X=2 X^=2 (4) Remember, 4 becomes the new value of X
%=	N/A	N/A	X=3 X%=2 (1) Remember, 1 becomes the new value of X
X++	N/A	N/A	Is equivalent to X+=1 Takes the value of X, adds 1 to it and then stores it in X again
X--	N/A	N/A	Is equivalent to X-=1
()	Parentheses	N/A	(5+5)*3 (30) Used to get around the order of operations - they work like normal mathematics. You may have any number of parentheses inside of other parentheses.

Inputs and Outputs

There are special types of variables that are called inputs and outputs. Every E2 that you write (until you get a LOT farther in this guide) will have at least one input or output.

Input

Inputs are variables that the values of are told to your E2 when it runs. You've used inputs before with other gates, but now you get to decide what they are. If you look at an 'add' chip with the wire tool out, you'll see an input called 'A', and if you right click, you can cycle through to other inputs it has too (the letters 'B' to 'H'), which the 'add' chip adds together. You can also look at a thruster, which has an input called 'A' also, which is the multiplier for it's force. Your E2 may also need to take in data to do things with it. Some of our early E2 examples will just be replications of existing wire chips.

Output

Outputs are variables that your E2 gives off, often so that other wiremod components can use them as inputs. For instance, going back to our first example, the 'add' chip, it has one output, the sum of 'A+B+C+D+E+F+G+H'. That value, you can use for a multitude of things, you could even wire the input ('A') of our aforementioned thruster to it, to make that it's force multiplier. Inputs are wired to outputs.

Take a minute and look at wiremod components - it's imperative that you understand what inputs and outputs are and how to tell the difference between them - if you don't, I can guarantee you that you won't be able to use E2. If the above text wasn't enough, there's a few things you can do, spawn components and pull out the advanced wire tool, when 'wire' components together, the first component you click on, you're getting an 'input' from, the component you wire it to, you're getting an 'output' from. Inputs go to outputs. If that still isn't enough, pull out the wire debugger and click on a component, it will list all inputs and outputs for that component, observe them, it should become clear. If you've never used the debugger before, know that you will become very familiar with it as we progress.

Making an Input or Output

Making an input or output is very easy. To do so, just go to the appropriate line (@inputs for inputs or @outputs for outputs) at the top of your E2 code and append (add to the end of it) the name of your variable. Thus, if I want to have an E2 with one input, 'X', and one output, 'Y', then I would have

```
@inputs X
@outputs Y
```

At the top of my E2. Note that you can have multiple inputs and outputs on each line, but you don't have to,

```
@inputs A B
```

is just as valid as

```
@inputs A
@inputs B
```

It's fairly obvious that you're not setting a value to your inputs or outputs on this line, so, then what do they hold? Well, the values that inputs hold is determined by what they're wired up to, outputs, you must set yourself. It's a good bet that by default they will be equal to '0'.

Our First E2s

Now that you understand inputs and outputs, you're capable of making basic E2s. I'm going to put the code here for some basic E2 gates that replicate existing ones, such as the add gate, and we'll examine each

line of code and figure out how they work. Before you read on, I have one more term for you:
Comments - A comment is usually an explanation of how something works that is written in the code itself, but has no bearing on what the code does, it's only there so that readers can better interpret it. Usually, you won't be commenting your code, because you won't be sharing it, however for the purposes of this guide, understand that comments are started with the ampersand character (#). Anything that follows an ampersand character on any line will not execute. In the E2 editor, everything that follows will be gray, here it won't be so obvious. Here are some examples:

```
#Here is an entire line devoted to a comment
X=5 #I set X to 5 and now I'm in a comment
#X=6
#In the above line, X is NOT set equal to 6, because that line is commented
```

Take a look at the second to last example for a moment. Notice that that's a valid line of code were it not for the # in front of it. Often, in the process of debugging (fixing your code), it will be a good idea to comment out some of your code to find out what works and what doesn't. Remember that for later.

Now for the first E2, we're going to replicate an Add gate, the one minor difference being that it will have only 2 inputs instead of about 8:

```
@name Add Gate #Remember, the name has no relevance to what the code does
@inputs A B #Variables start with capital letters and on the inputs/outputs line are separated by spaces
@outputs Sum
@persists
@trigger
Sum = A + B
```

Ok, so I commented most of the lines, but lets take a look at the very last one, the one that actually does something. We take the sum of 'A' and 'B' and set 'Sum' equal to that value. Because of that, if we wire, lets say, a thruster, to our E2, it will have the force multiplier of A + B (bear in mind A and B must be given to the E2 or they will be 0, as will Sum).

Simple, right? Well, it gets better, lets say you want to take A+B and then multiply it by 5, well, we can do that all in one E2.

```
@name Add Gate #Remember, the name has no relevance to what the code does
@inputs A B #Variables start with capital letters and on the inputs/outputs line are separated by spaces
@outputs Out
@persists
@trigger
Out = (A + B)*5
```

Ok, lets see what I did there - not a whole lot has changed. First, I added parentheses around 'A+B' to make sure it happens first, which is because the mathematical order of operations you were taught back in elementary school applies here, and lastly I added '*5' to the end, to multiply that whole value by '5'.

Try out the above examples and then read on.

Introduction to Functions

Functions are blocks of code which are designed to perform a single task.

The expression gate has many functions, not all of which will be explained in this guide and certainly not all of which will be covered in this section. For the time being, we're going to look at the functions listed in this section of the E2 wiki.

Speaking of the wiki, it's time to learn how to read it, take a look at the line, a little below the part of the page that the above link directs you to, that starts with 'sqrt(N)' on the left. This is a function that will tell you the square root of 'N'. I'll explain what N means and how to get the value of the square root in a minute, but for now, just understand that's what it does. There's a few parts to it, and we'll dissect them all:

- Starting on the left, we see 'sqrt(N)' - sqrt is the name of our function. All function names start with lower case letters. To call/run this function, you need to type 'sqrt' into your code.
  - Next we see a set of parentheses. Every function name will be followed by an open and then a close parenthesis, whether or not we put something in the middle is determined by the function definition (which in this case is: 'sqrt(N)')
  - 'N' is a parameter/argument. Parameters are pieces of data that are given to a function, similarly to an input, it's something that the function's code doesn't know before running, but rather, it gets it from someplace else. 'N' can be a variable or a constant ('5' for example), when you try and call/run the function. The reason I know it can be '5', as opposed to lets say "hello" (a string, something we'll learn about later), is because it's 'N' and not 'S'. 'N' always means numerical value (although there's no reason why in your code you couldn't have a string that was called N, this is just the wiki's notation).
- Next we see the 'returns' column. This column tells you what the function tells you. As I explained parameters as inputs, returns are outputs. A return is what the function gives back to you. The column says that 'sqrt' will return an 'N'.
- Finally, on the right, we have a description of the 'sqrt' function. It tells you that essentially, the return value is equal to the square root of the value that you passed to the function.

Calling a function - To call a function is to run it, to make it happen. You can call a function almost anywhere in your code and you can call the same function more than once. There's very few limits to where/when you can call them. Calling a function is a very simple thing to do. If I want to take the square root of '4', which you should know from school is equal to '2', then all I would type in my code is:

```
sqrt(4)
```

Now, I could do that, and it would work, but it's not very helpful, I'm not doing anything with the value it finds. In order to actually do something with that value, I could type something like:

```
X = sqrt(4)
```

And by the end of that line, X would always equal '2'. Now, I don't need to just put '4' in the middle of the parentheses as my argument/parameter, I can put another variable in there if I want or even do something like 'sqrt(3*3)', which will clearly return/give back, '3'. Functions can also be used in any operation I listed earlier too, here are some examples of valid uses of a function:

```
X = 5 + sqrt(4)
X = 3*(sqrt(4+6)-5)
```

One thing to note, is that functions are ran before other arithmetic operations take place, that means that if you type 'sqrt(2+2)*3', the multiplication will NOT run first, but rather, the addition inside the parentheses will run, then the function will find the square root of that sum, and then lastly, that value will be multiplied by 3.

Lets look at another function, the absolute value function. Scrolling down a little more on the wiki will reveal a function called 'abs('N')'. I shouldn't need to explain this one too much, so lets have a guess at what the following line does:

```
        X=abs(-3)
```

If you paid attention, and you know anything about mathematics, you should recognise that X has just been set to '3'.

Using that same premise will work for your understanding most of the functions listed in that section of the wiki I linked you to. I don't expect you'll know how to use all of them, you shouldn't either (especially if you don't know the math for them, trig, and logarithms, etc), but you should be able to figure out things like ceil and floor by their descriptions. Note, that almost every function (if not all of them) you see here are gates in the 'arithmetic-gate' tool of normal wiremod.

Understanding Booleans

Boolean statements are essentially logic statements. You'll find that every comparison we're going to make in this section can be done by a logic gate instead, however this way is more efficient in that it allows you to have more than one comparison done in the same gate. An example of a Boolean operation is '=='. '==' is different than '=' in that '==' is used to CHECK equality, whereas '=' is used to SET variables to values. It is common to forget about the double equals in your early code, which will cause you to have errors.
Boolean expressions always evaluate to be true or false, 1 or 0. Thus, if I type:

```
        X = 5 == 5
```

Then 'X' will be set to '1', because '5' is equal to '5'. Now, that does look confusing, so what I like to do, and what is standard in most programming languages, is to surround your Boolean expressions with parentheses like so:

```
        X = (5 == 5)
```

That's much less of a strain on the eyes for readers. They can tell instantly that the inside of the parentheses is a comparison.

You can also replace any numerical constant with another variable:

```
        Y = 3
        X = (Y == 5)
```

Which is clearly false, 0.

Lets say that we don't want to check equality, but some other comparison, well, we have operators for that also:

Operation	**Gate Equivalent**	**Example of Usage with the value of X (once the line finishes) listed after in parentheses**

==
Note: technically this is a numerical operator, so do not <u>assume</u> that it will work on other types (string/ vector/etc), even though it may.
	Equals	X = (5 == 5) (1)
		X = (5 == 4) (0)
		X = (3 == 9) (0)

!=
Note: technically this is a numerical operator, so do not <u>assume</u> that it will work on other types (string/ vector/etc), even though it may.
	Not Equal	X = (5 != 5) (0)
		X = (5 != 4) (1)
		X = (3 != 9) (1)

<
Note: technically this is a numerical operator, so do not <u>assume</u> that it will work on other types (string/ vector/etc), even though it may.
	Less than	X = (5 < 5) (0)
		X = (5 < 4) (0)
		X = (3 < 9) (1)

<=
Note: technically this is a numerical operator, so do not <u>assume</u> that it will work on other types (string/ vector/etc), even though it
	Less than or equal	X = (5 <= 5) (1)
		X = (5 <= 4) (0)
		X = (3 <= 9) (1)

may.		

>
Note: technically this is a numerical operator, so do not <u>assume</u> that it will work on other types (string/ vector/etc), even though it may.
	Greater than	X = (5 > 5) (0)
		X = (5 > 4) (1)
		X = (3 > 9) (0)

>=
Note: technically this is a numerical operator, so do not <u>assume</u> that it will work on other types (string/ vector/etc), even though it may.
	Greater than or equal	X = (5 >= 5) (1)
		X = (5 >= 4) (1)
		X = (3 >= 9) (0)

&	And	X = (5 < 10) & (3 > 2) (1)
		X = (5 < 10) & (3 < 2) (0)
		X = 1 & 1 (1)
		X = 1 & 0 (0)
		X = 0 & 0 (0)
		This operation is used to combine multiple Boolean expressions into one.

| (this character is made by using shift plus the key above enter)	Or	X = (5 < 10) | (3 > 2) (1)
		X = (5 < 10) | (3 < 2) (1)
		X = (5 > 10) | (3 < 2) (0)

!	Not	X = !(5 == 5) (0)
		X = !(5 == 4) (1)
		X = !(3 == 9) (1)

If Statements

If statements are practical applications of Boolean expressions (expressions is not referring to E2, '==' is called a Boolean expression). If statements allow you to make logical decisions in your code. For instance, in real life, I could say, if I'm in a store and I have $5 then I will buy a piece of candy, otherwise, I won't. Well, in code, you can do the same thing. An if statement looks something like this:

```
if(5==5)
{
    #Code here
}
else
{
    #Other code here
}
```

The first line of the if statement is 'if(5==5)'. If the Boolean expression contained within the parentheses (and note, the parentheses are NOT part of the Boolean expression in this example, they contain it, they are mandatory for an if statement) is true, then the first block of code, represented by the comment 'Code here' will run, or 'else', the second block will run, 'Other code here', but never will both blocks run. Blocks of code are distinguished by open and close curly braces ('{' and '}'). If you open a block, you should tab over for the code inside and keep each brace on its own line. The exception to that rule is when you only have one short line in your if, in which case it's accepted to keep the braces and the line on the same line as the if statement. While the E2 won't stop you if you decide to put things on different lines than I recommend, I recommend this practice because it's the standard. Every time you have an if statement you need to create at least the first block of code, the 'else' keyword and block are optional.

There is one more block involved with if statements, the 'elseif' block. This allows you to have multiple if statements written in the same if statement. Now you may be wondering why this is useful, well, there are a few reasons, one, is that the 'else' block only happens if none of the above conditions are true. Two, is that you can check for two conditions that might both be true, however you only want one of them to execute their code, not both, for instance:

```
X = 1
if(X < 5)
{
    #Code here
}
elseif(X < 10)
{
    #Other code here
}
```

You can tell that 'X' is definitely less than '5' and that it's also definitely less than '10', however when we run this code it will only run the 'Code here' block, not the 'Other code here' block, because in an if statement, only one block ever runs. If you wanted to run both blocks, you would make the 'elseif' block into it's own if statement, rather than have it be part of this one. You'll also notice that I do not have an else block here, as I mentioned earlier, it's not needed. The only mandatory block is the first block, the if block.

You can also use the '&' and '|' operators (previously explained) in ifs, for instance:

```
X=1
Y=2
if(X == 1 | Y == 2)
{
    #Code here
}
```

Will cause the shown block to run. Keep in mind that parentheses used with the '&' and '|' operations can be very powerful tools in changing the meaning of your Boolean expression.

The '?' Operator

While it's less commonly used than if statements, it's just as an important use of Boolean expressions as they are. The advantage to the '?' operator is that if you're only going to create an if statement to set a variable to one value if the condition is true and another if it's not, then the '?' operator is what you should use because it evaluates faster. Here's an example:

```
X = (5 < 3 ? 25 : 53)
```

What this does, is it checks if '5' is less than '3', we know that this is false, and so it looks for the false value of the '?' expression, which is the value after the colon (:), in this case, '53', and then sets 'X' to that value. If the Boolean expression was true, then it would set it equal to '25', the value directly after the '?'. In short:

```
(Boolean is true ? then this : else this)
or
(Boolean ? true : false)
```

You'll most likely end up using this primarily to set the values of variables, however you can use it inside of if statements, functions, or even inside of other '?' operator expressions. Bear in mind that if the value you want for true is '1' and the value you want for false is '0' that you should just use the Boolean expression alone, and not include the '?' operator, it will do the same thing.

Cycles/Timers/Persists

Cycle - A cycle is an execution of your code. Basically, every time your code runs from the top of your E2 to the bottom of your E2, you've just finished a cycle. By default, your E2s will run 1 cycle and then stop. You can make it so that they run more cycles in various ways:

- You can alter one of the inputs - every time an input changes, your E2 will run (by default this is true, but you can alter this functionality with the '@trigger' line, as we'll learn in the next section)
- You can use special commands that cause one E2 to cause another E2 to run

- You can set timers that cause the E2 to run after a certain amount of seconds

Timers are an essential part of most expressions. The most commonly used one is called an interval. Interval causes your E2 to have a cycle every 'N' number of milliseconds. To create an interval, just add the line:

```
interval(N)
```

to your E2. It's common practice to put it as the first line below your '@trigger' line, however it's not necessary to do so. It's also not necessary to have it be a continuing interval, you can include the line in an if statement if you like. One thing to know is that 'N' cannot be lower than '10' (if it is it will just change to '10' anyway), and it shouldn't be lower than '200' unless you have a good reason.

Less commonly used is the 'timer' function, which causes your E2 to run after the given number of milliseconds, like interval. The difference is that you can have more than one and that it can have a name to refer to it as. The name lets you check if that was the thing that started the current cycle of the E2 by using the 'clk' function. Here's an example of a timer:

```
timer("test", 100)
if(clk("test"))
{
    #The timer 'test' has caused the E2 to run
}
```

It's important to note the quotes (") around the word 'test', this tells the E2 that 'test' is what's known as a string. We'll learn more about strings in a short while, but for now, understand that it signifies text. Also, note that 'clk' can be used without a parameter to tell if the cycle was started by an interval.

Early into this guide I mentioned the '@persist' line. Well, now that we know more about cycles, we can actually use it. A persistent variable is one which keeps its value between cycles. Most variables, when the E2 goes from one cycle to the next, are cleared and lose their value. Inputs, outputs, and persists, however, keep their values between cycles (assuming an input doesn't change due to outside forces). Thus, if you would like to have a variable that isn't an input or output and you need it to keep it's value (possibly to change over time), then you should list it on the '@persists' line. Here's a good example of a use for that functionality:

```
@name Simple Timer
@inputs On
@outputs Value
@persist Time
@trigger
interval(1000)
Time++
        Value = (Time % 2 == 0)
```
Ok, lets take a look at this. What does it do? It sets 'Value' equal to '1' every '2' seconds. We have an interval of '1000', meaning that it runs once every second, and when it does so, the value 'Time' increases by '1' due to our '++' operator. We then check the value of time to see if it evenly divides by '2', and if it does, 'Value' is set to '1', otherwise it's set to '0'.

The Trigger Directive

The @trigger line at the top of your E2 determines what inputs, if changed, will cause your E2 to start a new cycle. By default, all of them will. You can, for redundency's sake, type:
```
        @trigger all
```
To make your E2 run whenever an input changes (in addition to timer changes).

If you want only certain variables to trigger a new cycle, then you just list them, for instance:
```
        @inputs X Y
        @outputs Z
        @persists
        @trigger X
```
Will make a new cycle whenever 'X' changes or a timer occurs, but NOT when 'Y' changes.

If you want to override the default functionality such that only timers will cause a new cycle to begin, then you can type:
```
        @trigger none
```
On your trigger line.

Timers will always cause your E2 to execute, if you want to remove a timer or stop it from executing constantly, see the 'stoptimer' function on the wiki, or just place your timer in an if statement.

String Variables

String variables are more or less used the same way as numerical ones. They store data, in this case, text data such as "hello", or "Avast ye land lubber!". They're used in a whole bunch of functions, and the parameters/returns for them are indicated by an 'S'.

Creating a string variable is different then creating a numerical one in some instances. If you're just creating it in the middle of your code (ie: it's not an input/output/persist), then you can do the same thing as normal, for instance:
```
        X = "hello"
```
However, if it is an input/output/persist, then you need to do something a little more complicated. Lets say you need a string input called 'X', then you inputs line would read:
```
        @inputs X:string
```
From there, you can use 'X' and set it or get it's value like any numerical variable. If you want to get an input of a string, you can do so from any other wire component, including expression gates, to find out if a component has a string output, use the advanced wiring tool to look at the outputs of a component and determine if it has a string output.

String operations are also different then numerical ones. While some are the same, many of them are no longer present:

Operator	**Example of usage with the result in parentheses**
+	X="hello " + "there" ("hello there")

+=	X="hello "
	X+="there" ("hello there")

==	("hello"=="hello") (1) ("hello"=="yarr") (0)
!=	("hello"=="hello") (0) ("hello"=="yarr") (1)
~	This is used with inputs to see if they changed. It will be '1' if the input changed, '0' if it did not. ~X

Understand that strings are unlikely to be a part of your program unless you're specifically trying to deal with text, something which doesn't happen usually until you get into the advanced sections of this guide. I explain this here because it can become useful with entity manipulation, which will be explained a few sections down.

One thing to keep in mind when you use the trigger directive, is unlike the '@inputs' and '@outputs' lines for variable types other than numerical, you do not need to type the name of the variable and it's variable type. IE: If you have a string 'X' as an input, and you want it to be a trigger, typing '@trigger X' is proper syntax, you do not need to type '@trigger X:string'.

Functions (Continued)

Some functions are listed with the prefix 'Type:', in this case, 'S:' on the wiki, this means that you're running that function on a variable. The best way to explain how this works is by example:
```
                X = "hello"
```

```
X = X:upper()
```

At the end of this example, 'X' is "HELLO". We just ran the 'upper' function on 'X'. Essentially, 'X' is a parameter to the function, we just listed it in a different manor. Most functions for non-numerical types will use this syntax.

Vector Variables

Vectors represent X, Y, Z coordinates. Generally, you'll be using valid coordinates in your map, but on occasion you may find yourself entering the coordinates by hand. Vectors can be considered 3 numerical variables in one. Lets examine the wiki on vectors:

First things first, ignore all but the '3D Vector' and 'Common Vector' subsections. The other two are not commonly used. Second, lets look at how to make vectors. It's fairly simple, you need only type 'X = vec(0,0,0)' to set X equal a vector of point 0, 0, 0. You can also input/output vectors, similarly to strings, you need only append the name of the type after a colon (:) after listing the variable name on the line:

```
@inputs X:vector
```

Creates the input 'X' and tells the E2 that it's a vector. If you want to get an input of a vector, you can do so from any other wire component, including expression gates, to find out if a component has a vector output, use the advanced wiring tool to look at the outputs of a component and determine if it has a vector output.
If you hadn't guessed by now, 'V' represents vectors on the wiki.

Functions work the same way for vectors as they do for strings and numerical variables.

Vector operations:

Operator	**Example of usage with the result in parentheses**
+	X=vec(0,0,0) + vec(90,0,0) (90,0,0)
==	(vec(0,0,0) == vec(0,0,0)) (1) (vec(0,5,0) == vec(0,0,0)) (0)
!=	(vec(0,0,0) != vec(0,0,0)) (0) (vec(0,5,0) != vec(0,0,0)) (1)
~	This is used with inputs to see if they changed. It will be '1' if the input changed, '0' if it did not. ~X

Angle Variables

Angles represent pitch, yaw, and roll. Generally angles are used with entities and thrusters to stabilize contraptions, but there are plenty of other valid uses. Like vectors, angles contain 3 numerical variables in one. Like vectors and strings, you need to put the type of the variable (angle) after it's name and a colon (:) if you declare it as an input/output/persist. Functions work the same way as they do always. For reference, 'A' represents angles on the wiki.

Angle operations:

Operator	**Example of usage with the result in parentheses**
+	X=ang(0,0,0) + ang(90,0,0) (90,0,0)
==	(ang(0,0,0) == ang(0,0,0)) (1) (ang(0,5,0) == ang(0,0,0)) (0)
!=	(ang(0,0,0) != ang(0,0,0)) (0) (ang(0,5,0) != ang(0,0,0)) (1)
~	This is used with inputs to see if they changed. It will be '1' if the input changed, '0' if it did not. ~X

Entity Variables

Entities are perhaps most central element of E2s. They're what all of the other types revolve around. Entities are props, players, ragdolls, and just about any object in Garrysmod. If you can find it with a target finder, it's an entity. You get angles and vectors from entities and use them in your expressions, often in conjunction with thrusters. Like vectors, angles, and strings, you need to put the type of the variable (entity) after it's name and a colon (:) if you declare it as an input/output/persist. Functions work the same way as they do always. For reference, 'E' represents entities on the wiki.

Entity operations:

Operator	**Example of usage with the result in parentheses**
==	Nearly impossible to show in a simple example, but use your imagination.
!=	Nearly impossible to show in a simple example, but use your imagination.

Appendix A

The '@' directive lines can be split onto multiple lines. In short, that means you can have more than one '@inputs' line or any of the other lines that start with '@':

```
@inputs X
@inputs Y Z
```

Is completely valid. I advise, however, that you avoid this unless you're:

1. Using a ton of variables (and then I advise you make sure you need all of those variables first)
2. You're organizing variables by type/purpose (and even then you'd need to have quite a few variables to make it worth it)

Appendix B

One of the most commonly used commands in E2 is the 'entity()' command. This returns the entity of the E2 itself, allowing you to use it as you would any other entity. In using this, just be mindful of where you put your E2 if you're going to try and find it's position or angles.

Appendix C

This appendix is in regards to debugging. I will most definitely add more to it later, but for now this will have to suffice. If you have an error when you try and press the upload/exit or save/exit button, then it's pretty easy to fix usually, you probably made a typo and it'll tell you the line where you did; however, if you have an error mid-execution or the E2 doesn't do what you think it should, you may need to debug more manually. By this, I mean you need to create outputs which indicate where your problem is. Why, you may wonder, well, the reason is because we can use the debugger tool on the E2. The debugger, I believe I already mentioned but I'll reiterate, shows you all of the values of your inputs and outputs. If we make specific outputs, we can determine where flaws in our code are.

The easiest thing to do first is to take any persistent variable and cut/paste it to become an output. If that doesn't work, adding an output, lets say, 'Moo', and running the line 'Moo=1' at one point in your code and 'Moo=2' at another, can show you in your debugger if an area is being ran. If you want to know if one specific section is being ran, then line such as 'Moo = (Moo == 1 ? 0 : 1) may be more appropriate.

Exploration

At this point in the guide, I must ask you to take the above knowledge and use it to explore the various functions of entities, angles, vectors, strings, and numerical variables. I ask that you take contraptions that you make using gates, and translate them as best you can into E2 code. The purpose of this, is to familiarize yourself with the E2. As of this point, you know the basics of the E2, it's time to use them. I will later modify this guide to include activities and projects for you to try, however for the time being, make them yourself. I will write additional material regarding the E2's complex features in the future in a separate guide.