# HCA Tech Note 103

## Expressions

This technical note provides several examples on some of the common uses of expressions and the Compute element. The Compute element opens a lower level of HCA than available from the Visual Programmer but requires a greater depth of understanding with concepts from more traditional programming languages. But once mastered, a great deal of additional function can be created in your automation solution using expressions.

This technical note is not a tutorial on programs or expressions in general. These references provide the background you might need. These can all be found on the support web site.

- User Guide chapter 11: Programs
- User guide chapter 13: Expressions
- Tech Note 107: Programs with Parameters

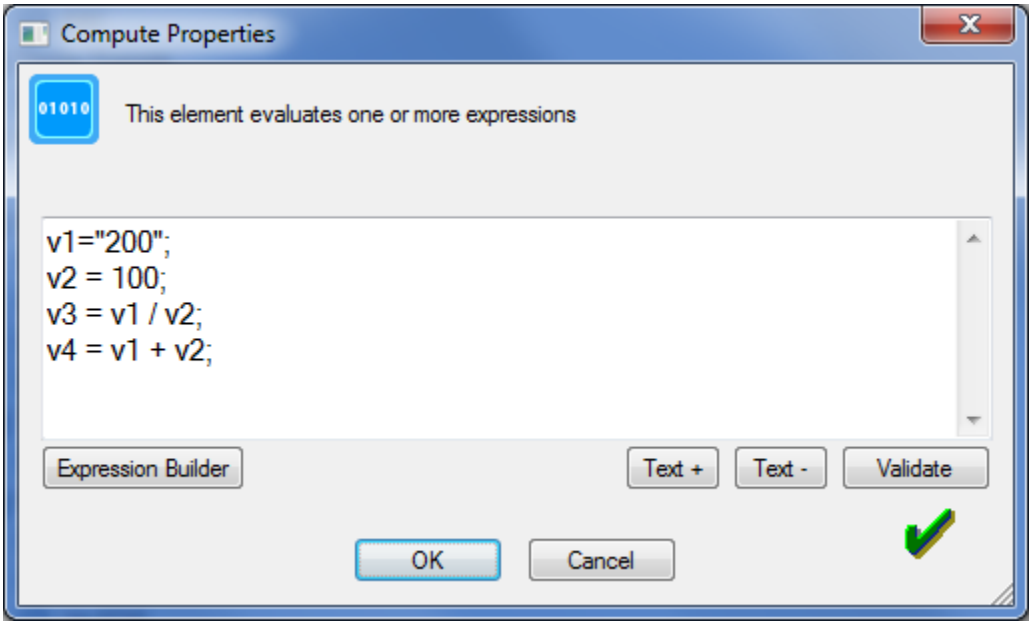
In addition to the general reference material, two of the examples use other HCA features. One uses the HCA Keypad and the other uses the Alexa facilities for starting programs. These are documented here:

- User Guide chapter 18: HCA keypads
- Tech Note 100: Programs that operate as devices

If you have not encountered HCA expressions in much detail you might see a similarity to VBScript. This is intentional. Not all the functions available in VBScript are in HCA expressions nor is, of course, the control structures. But some of the online tutorials and references may be helpful.

## Example: Conversion

To get started with a very simple example and to get used to the Compute element let's do something easy but showing one of the complexities of expressions. Here is a Compute element:




The first thing to note is that the text contains a series of assignment statements. On the left of the "=" is a name of a variable (called for historical reasons in HCA a "flag"). Each statement is separated from others by a ";".

If you are used to traditional programming languages, you will need to adjust to a few things that might not be familiar. The first is that variables are not declared. Variables come into existence when an expression is executed. The second major point is that variables can contain values of any type – a string, number, Boolean value (yes/no, true/false) or a date-time. When a function or operator does something with the value of the variable, if it is not the expected type an attempt is made to convert it into the correct type before use. This example shows this.

The first expression in the above Compute element assigns a string containing "200" to v1. The second statement assigns the number 100 to the v2 variable.

The next statement divides the contents of v1 by v2. Since the division operator requires two numbers to operate on, it converts the string "200" into the number 200 and then performs the division yielding – hopefully – the value 2.

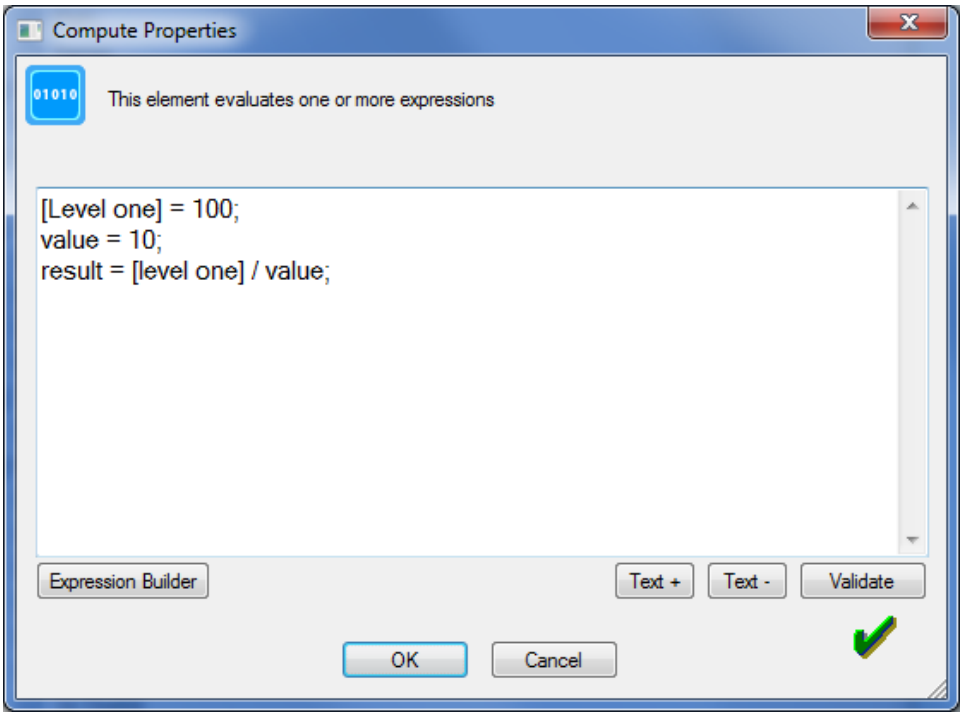
The next statement assigning a value to v4 looks simple but has a complexity. While the "+" operator usually means addition, it can also mean string concatenation (a programmer word for gluing two strings together). In this case the "+" operator operates differently. If either side of the "+" is a string then it makes a string out both sides rather than a number. This results in V4 being assigned the string value "200100". Perhaps not expected but it makes it really easy to do things like:

Text = "The light is at a level of " + LightLevel.

As you work with expressions and variables, you should use the tool that shows all variables and their current contents. It is called the "Flag Inventory" and there is a button to open it from the *Control* ribbon category, *Tools* panel, *Flags* button.

One nice feature of the Flags Inventory is that you can leave it open while you do other things and it automatically updates as variables change their values. It also has facilities for changing a variable's current value and seeing in what programs the variable is used. When working with expressions it will become your close friend. Make its acquaintance.

A few final words about variables. The variable name can contain spaces but then you must enclose the variable name in square brackets when using it in compute elements. For example:

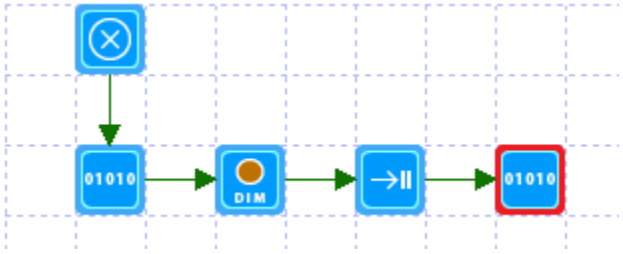



Also in this example you can see that variables have case insensitive names. In the first statement the variable is named "Level one". In the last statement it is named "level one". In the first with a capital L and in the second use with a lowercase l. It makes no difference as both refer to the same variable.

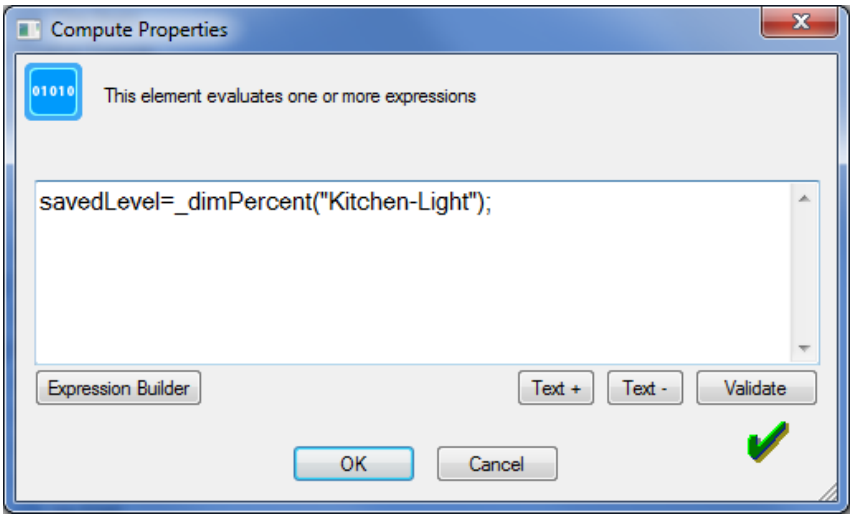
## Example: Level Restore

The next example solves this common problem: Capture the state of a device – it's level – and then increase the level for some period of time then restore the device to the level it was at.

The program looks like this:

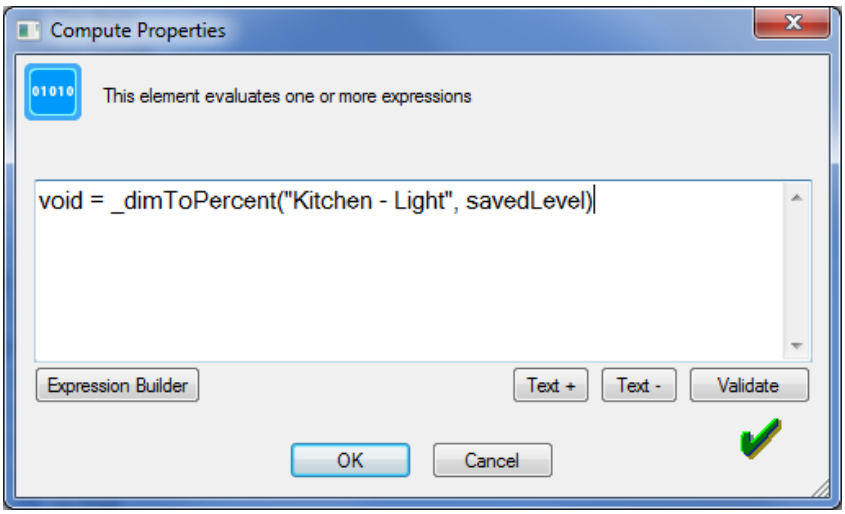



The key is in the two Compute elements. In the first is this expression.




The _DimPercent function returns the level a device is at. This saves the device level before being changed for use later.

The second Compute element has this expression.



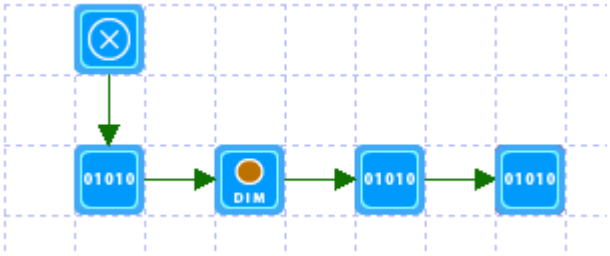
What's up with the "void" variable? All HCA built-in functions retyurn a value and the syntax of an expression requires something on the left and right sides of the equal sign. I use "void" as the name of the variable I assign to when I don't care what the result is. I could have used "x" or "ignore" as well but I use "void". The bottom line is that you must use something when you use a built-in function like this you need some variable to assign the result to even if you don't care what it is.

And that's all there is to it for this simple example. The program starts, captures the light level, increases it to 100%, delays, and then restores the level to what it was before the increase.

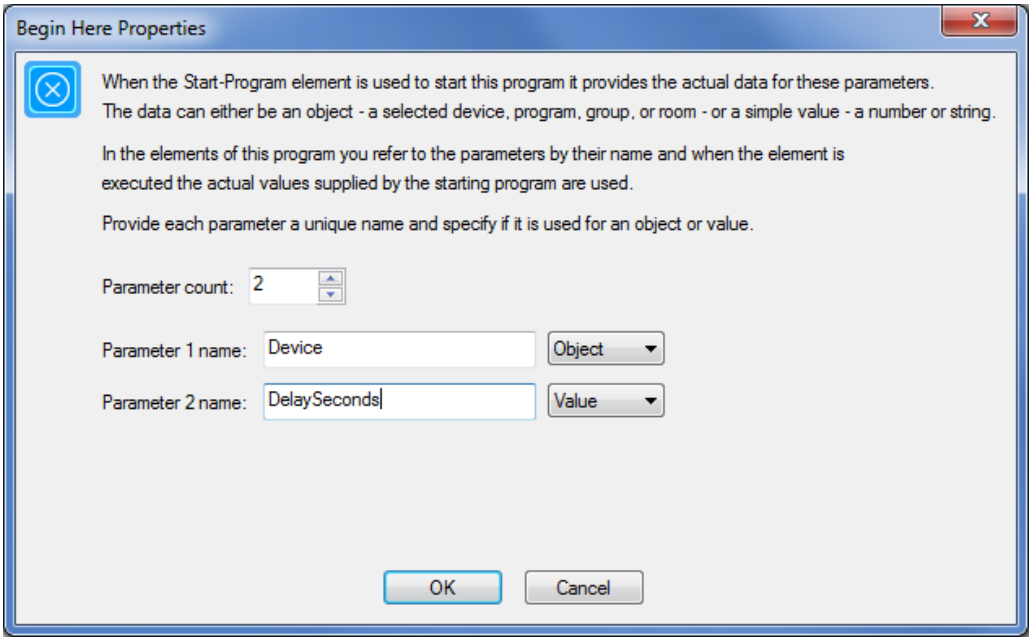
This is the first example that shows the use of a HCA function. All built-in functions start with an underscore character. The User Guide expressions chapter has the details of all functions and is a good reference. When creating expressions, you can also use the Expression Builder which is a tool to compose expressions and in particular works with the built-in functions. It is opened from the Compute and Compute test dialogs from the *Expression Builder* button.

While this solution works as expected it isn't as useful as it could be. The problem with this example is that the program only works on the one device "Kitchen – Light" and the delay element always delays for the same period of time.

A better program would take parameters which supplies the device to control and the number of seconds to delay. This program looks like this:

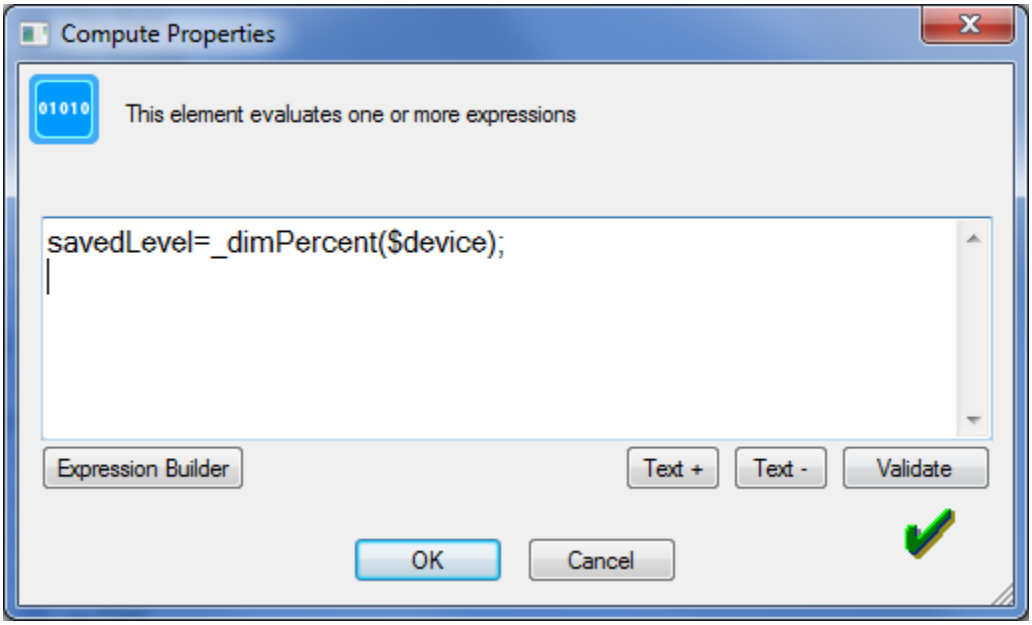



When a program accepts parameters, they must be defined in the Start-Here element:



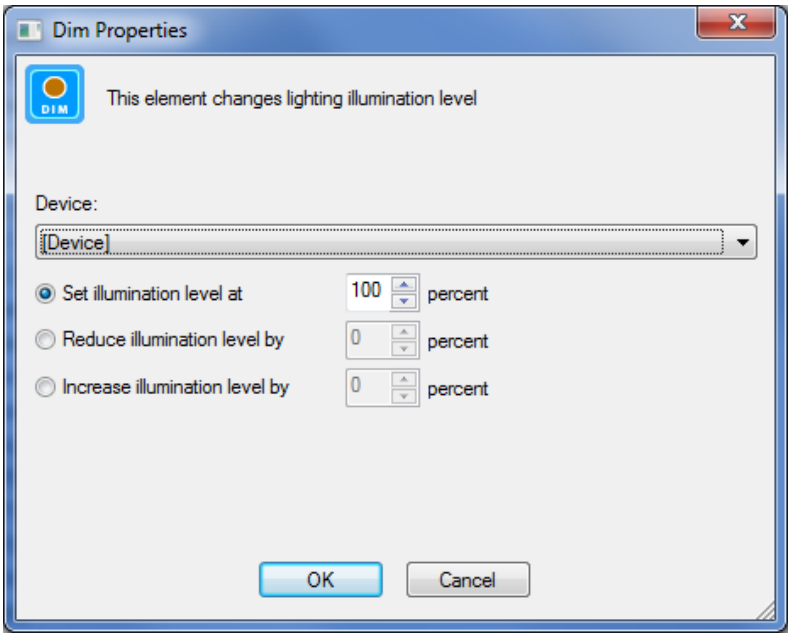

The program takes two parameters the first is the device to operate on and the second is the number of seconds to delay.

The first Compute element is like before but instead references the parameter.



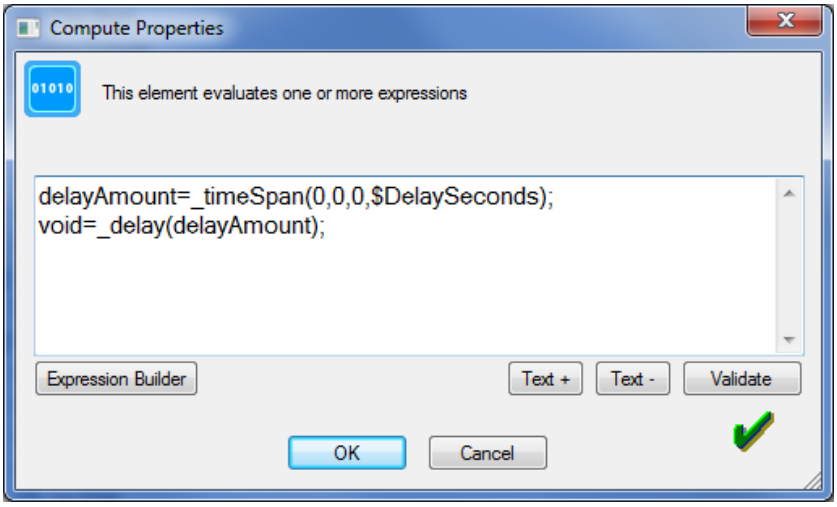

In expressions parameters are referenced by their name preceded by a "$" character so HCA knows that it is the name of a parameter rather than a variable.

The Dim element also is changed to reference a parameter:




When a program has “object” parameters they become choices in elements that operate upon devices, programs, and groups. In this example, the device to operate upon is called “Device”.

The next Compute element in the old program was a Delay element. In this new program that delay is done using a Compute element with a delay expression.

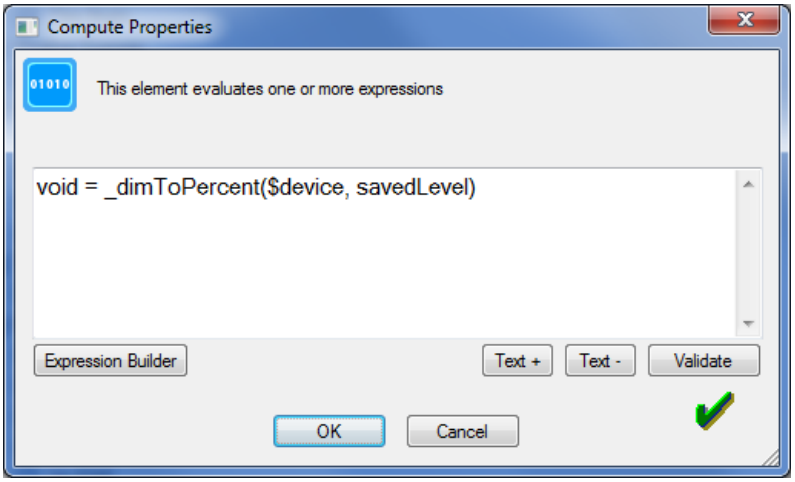



In expressions, there are variables which represent a time and date (Oct-2-2016 6:05:00 PM) and there are other values that represent a span of time (2 minutes 10 seconds). There are different expression functions that work with times and

others that work with time spans. In this case the *_TimeSpan* function is assigning to the variable *delayAmount* a span representing 0 days, 0 hours, 0 minutes, and the number of seconds passed into the program as a parameter.

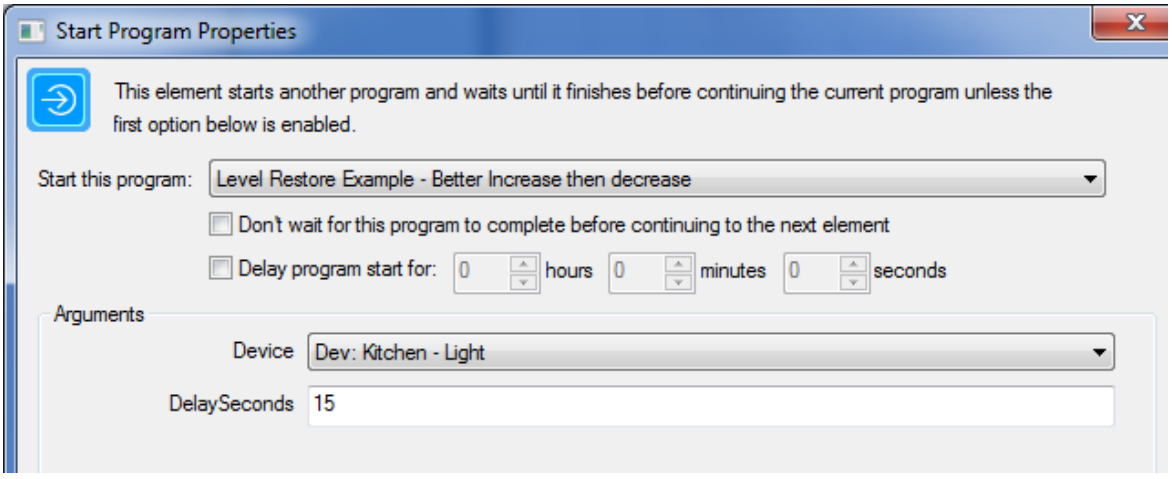
As a helpful note, with a time span you don't have to use 0-24 for hours, 0-60 for minutes, and 0-60 for seconds. You can create a span of one and a half hours (90 minutes) by: _TimeSpan(0, 0, 90, 0).

The last compute element is similarly modified from the simpler program:

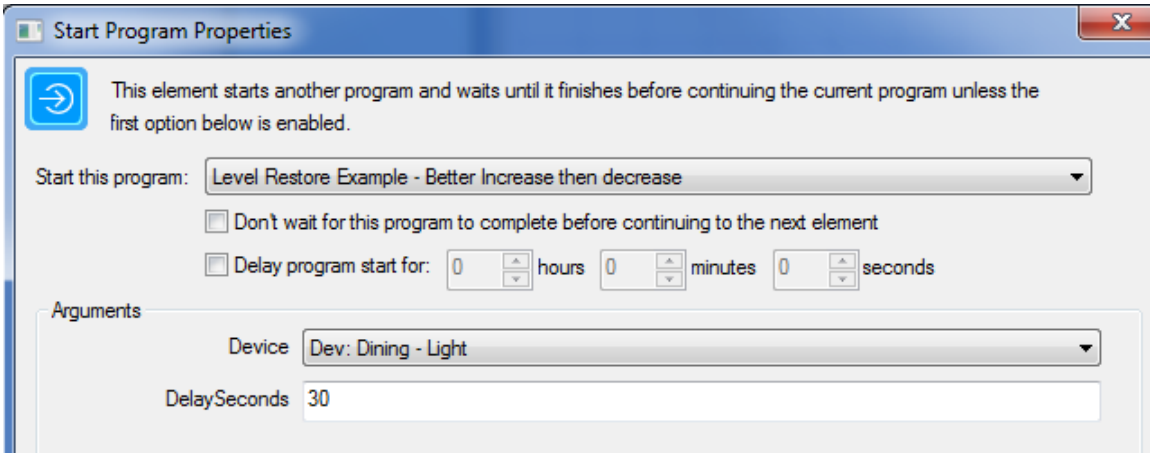



Again, the device parameter is referenced rather than a specific device chosen.

Now that this new program was created how is it used? In another program you can add a Start-Program element with its properties set like this:




Another use could be this:

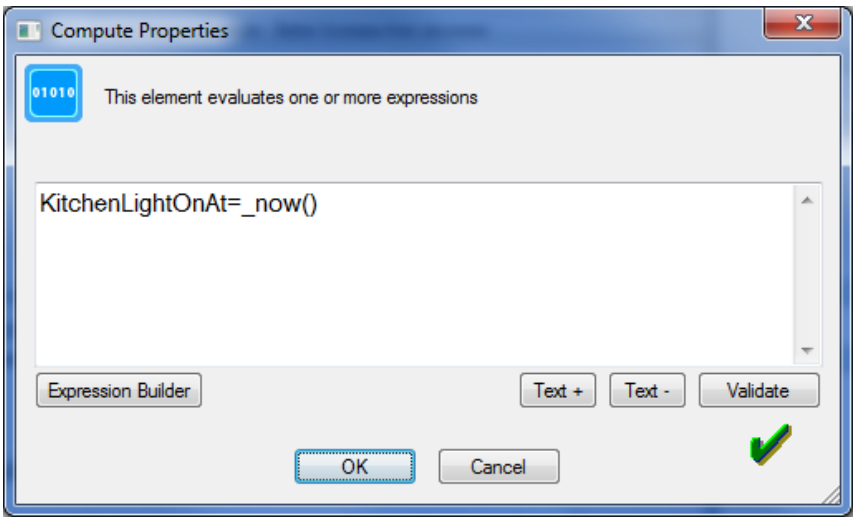



Note in the first case the program operates upon “Kitchen-Light” for 15 seconds, and in the second use that same program operates on “Dining – Light” for 30 seconds.

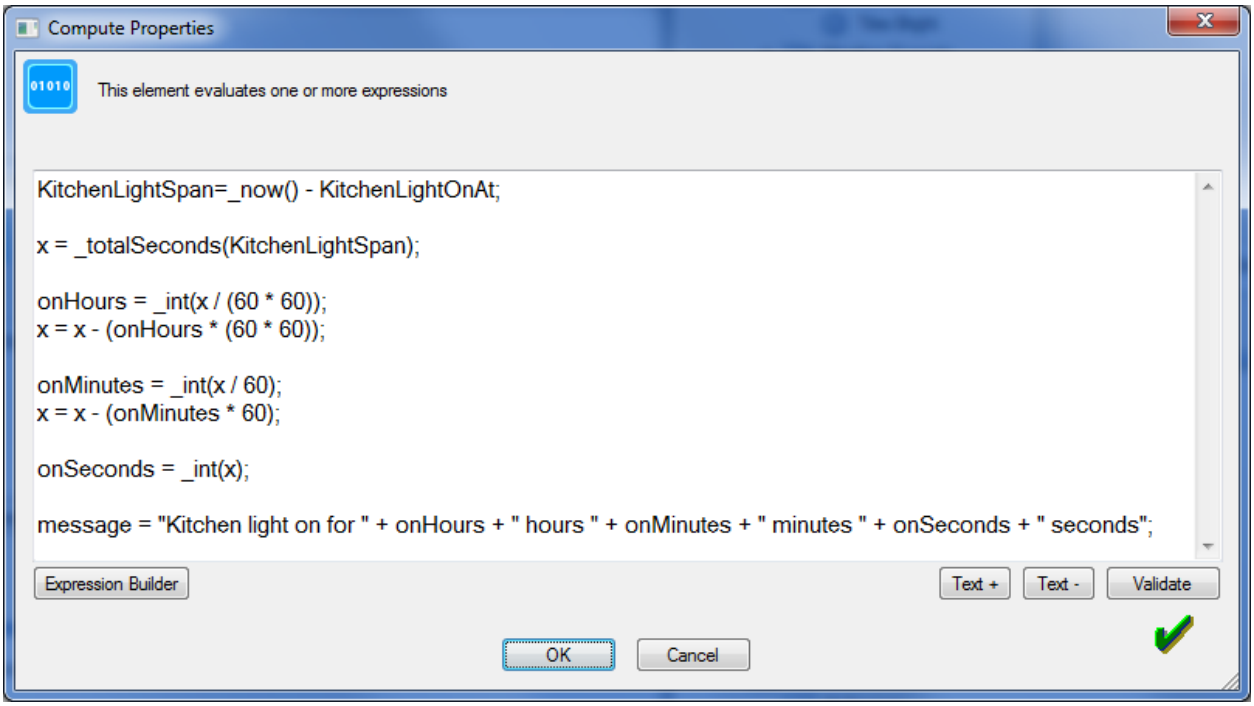
There is one big caution: In HCA all variables are global so if a program starts this increase-and-restore program and then while the delay is running another program also starts it then contents of the “saved Level” variable will get overwritten. There is no way to make a variable “local” to a program as in more traditional programming languages.

## Example: Timing

This next example works more with time values. The program use the “state change” trigger to have it started when the Kitchen-Light goes on or off. The program starts with a Test element to see how the program was started – when the light goes from “on to off”, or the light goes from “off to on”. The off-to-on path executes this Compute element:



All it does is to save in a variable the time the light went on. In the On-To-Off path, this Compute element is executed.



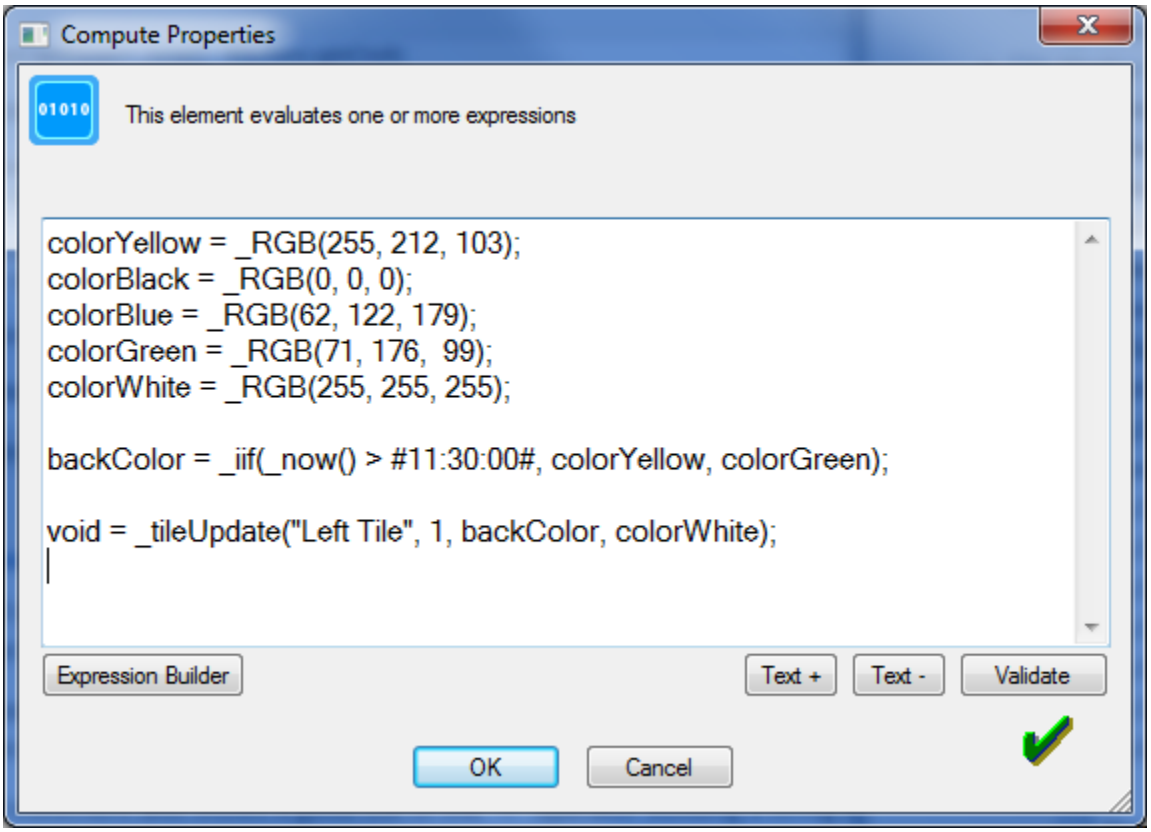

The first statement computes a span of time from when the light was first turned on until it was turned off. The next statements turn that into the number of hours, minutes, and seconds.

Note in this example the use of the *_int* function. In HCA numbers are always floating point numbers and the method used to convert the time span into house, minutes, and seconds needs to work with integer values to make the match work out. The _int function removes the fractional part of the number. If it still seems odd, work though an example to see how it works.

The last statement also shows a good use of the "+" operator building a string for display. The onHours, onMinutes, and onSeconds are all converted from numbers to strings as the final result is assembled.

## Example: Display

In this example the idea is to change the background color of a tile on a display depending upon the time of day. Here is the Compute element:

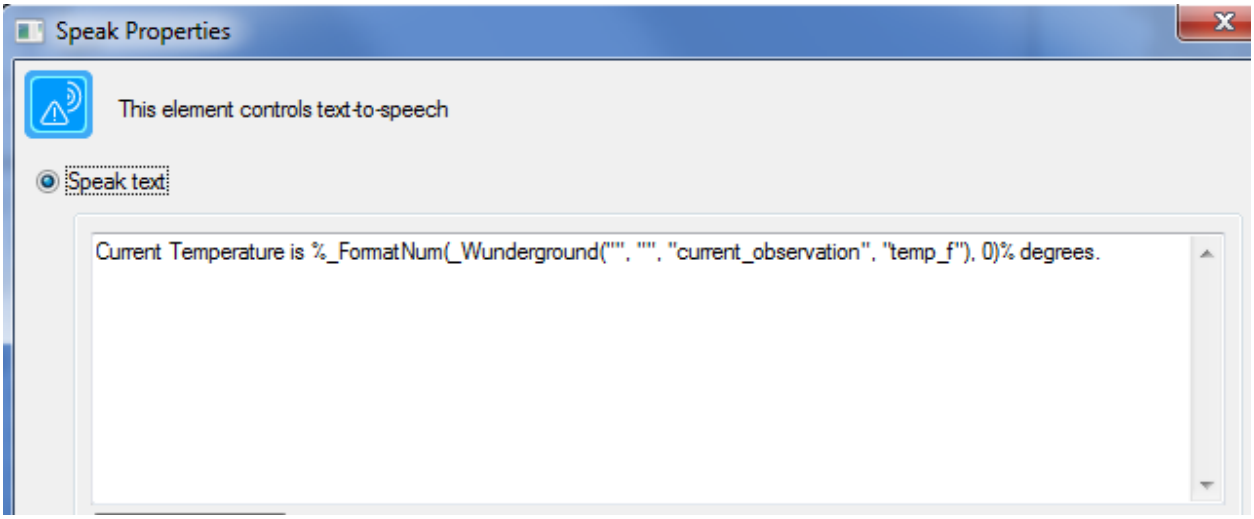



Most of the statements are similar to ones you have seen before but there are a few things to note. The *_RGB* function makes a color from red, green, blue. The assignment to the backColor variable uses the *_iif* function. This is a handy method to test the value of something and if the test succeeds to use one value and a different value if the test fails. In this case the test sees if the time is after 11:30 in the morning and chooses one of two colors.

You can embed dates and time within #'s and HCA converts them into a date-time. Be careful with this as the format of the string you give it, while open to many possibilities, isn't infinite. It also uses the computer's local settings to determine things like date ordering (month-day-year or day-month-year for example).

## Example: Weather

Expressions can also be used outside the Compute and Compute-Test elements. In other elements that operate on text (Show-Message, Speak, etc) you can embed expressions in the text surrounded by percent signs. When the element executes, any expressions are evaluated and their result turned into text and the final text then is used by the element. For example:



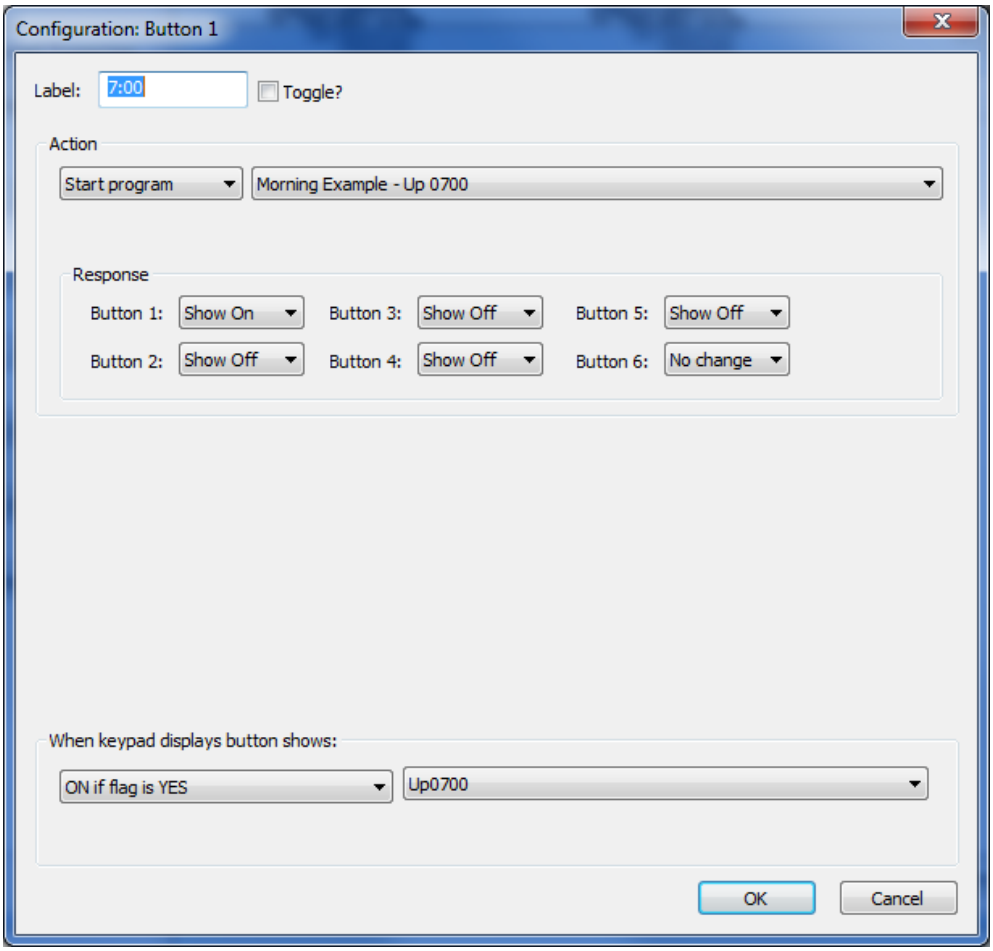

In this example the text-to-speech will say “Current temperature is” followed by a number and then the word “degrees”.

The embedded expression first uses the _Wunderground function to get the temperature in degrees Fahrenheit. Unfortunately, Weather Underground provides the temperature with a tenth of degree precision. For this use that level of precision isn’t needed so the value is passed into the _FormatNum function with returns a text representation of the number to a specified number of decimal places – in this case with zero decimal places.

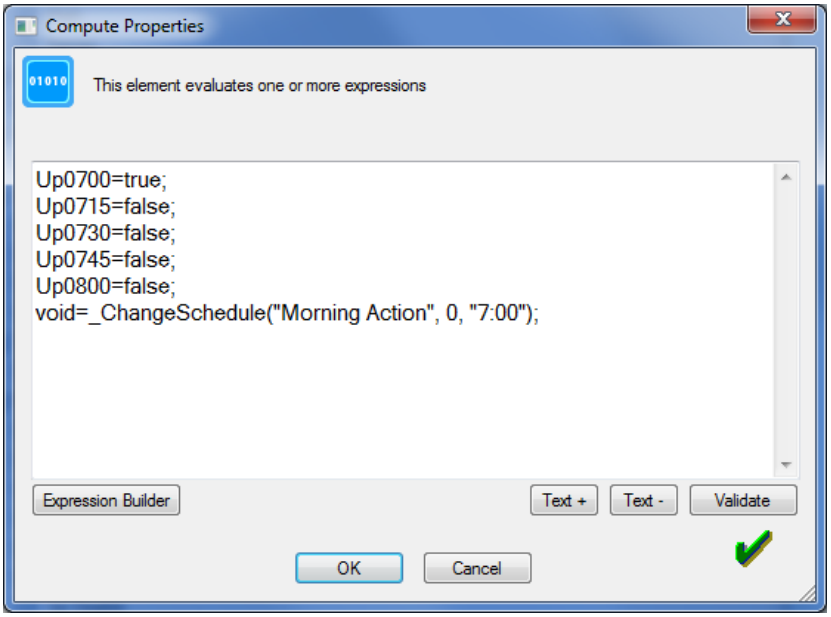
## Example: Morning

This next example is a bit bigger than the others. A virtual keypad has been created called “Home – Keypad”. The idea is that it could be displayed on a mobile device and used to change a schedule entry for what time to execute a “wake up” program.

Each button of the keypad is configured to run a program and to also use a variable to show if the button appears as “on” (yellow) or “off“ (gray) when the keypad appears.  Here is the configuration of just one button. The others are similar.




The “Up 0700” program executed when the button is pressed contains one compute element.




The Compute element has a series of expressions that assign values to variables as needed to make the keypad buttons display as ON or OFF. The final statement is where the real work is done. The *_ChangeSchedule* element takes three arguments. The 1st is the name of a schedule entry, the second is a code for what part of the entry to change, and the last argument is what to change it to. In this case the ON time is being modified.

The last button on the keypad enables or disables the schedule entry so you can kill the schedule for the next day and sleep late.

The key piece of this example is how the keypads interact with programs, variables, and in this case a schedule entry that is modified as needed.

## Example: Alexa

This final example is the most complex yet. The idea is that there is a light – "Downstairs Lights" and in Alexa we would like to be able to say these things:

*Alexa, turn on downstairs light*

*Alexa, turn off downstairs light*

*Alexa, increase downstairs light by 10 percent*

*Alexa, decrease downstairs light by 20 percent*

*Alexa, dim downstairs light*

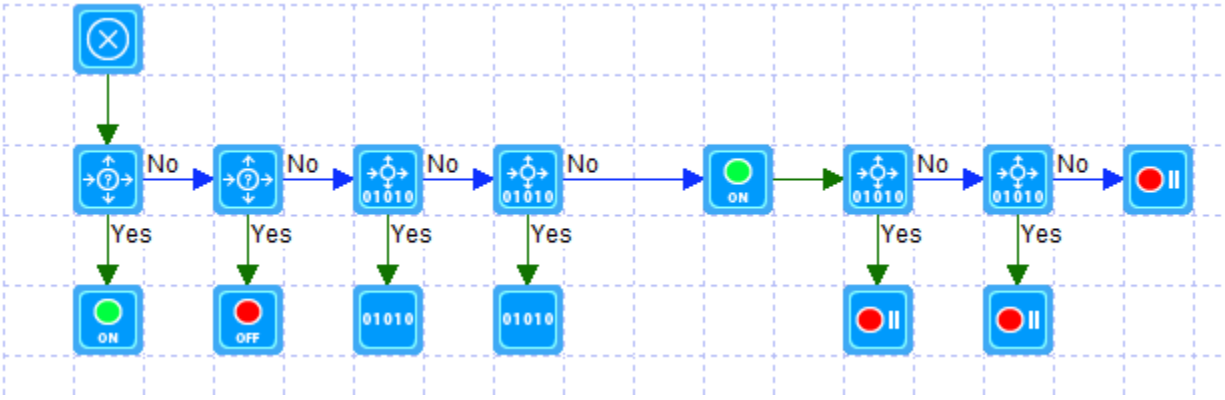
*Alexa, brighten downstairs light*

*Alexa, set downstairs light to 10*

The first six in this list are the usual control for lights and HCA handles all of that. But the trick is in the last one. In this case rather than setting the light to some illumination percentage the goal is to turn the light on and set an auto off timer appropriate for what was said: 10 minutes, 20 minutes, or 30 minutes.

There are several ways of doing this. One would be to have HCA handle all but the last case and work with the light directly. Then create new programs and have Alexa run those – like "Lights ten", "Lights twenty" or "Lights thirty".

Rather than that approach, let's create a program called "Downstairs – Light" and have Alexa control it and that program control the actual light. This let's us use may of the features learned in the other examples. The program looks like this:




The program has a generic ON and a generic OFF trigger and two parameters. This technical note explains those parameters:

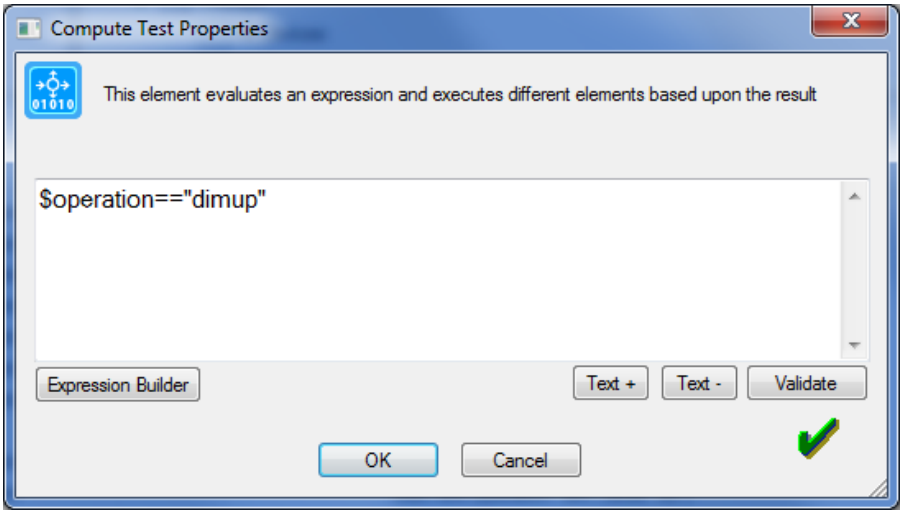
http://www.hcatech.com/download/Doc/TechNote_AlexaChangesForPrograms.pdf

As it says in there, all the Alexa phrases for dimming and brightening can apply to programs as well. In this case the program receives info as to what is happening by the first parameter – a string of either "dim", "dimup", or "dimdown". The second parameter is the level to dim to or the amount to increase or decrease. The program can then test these.

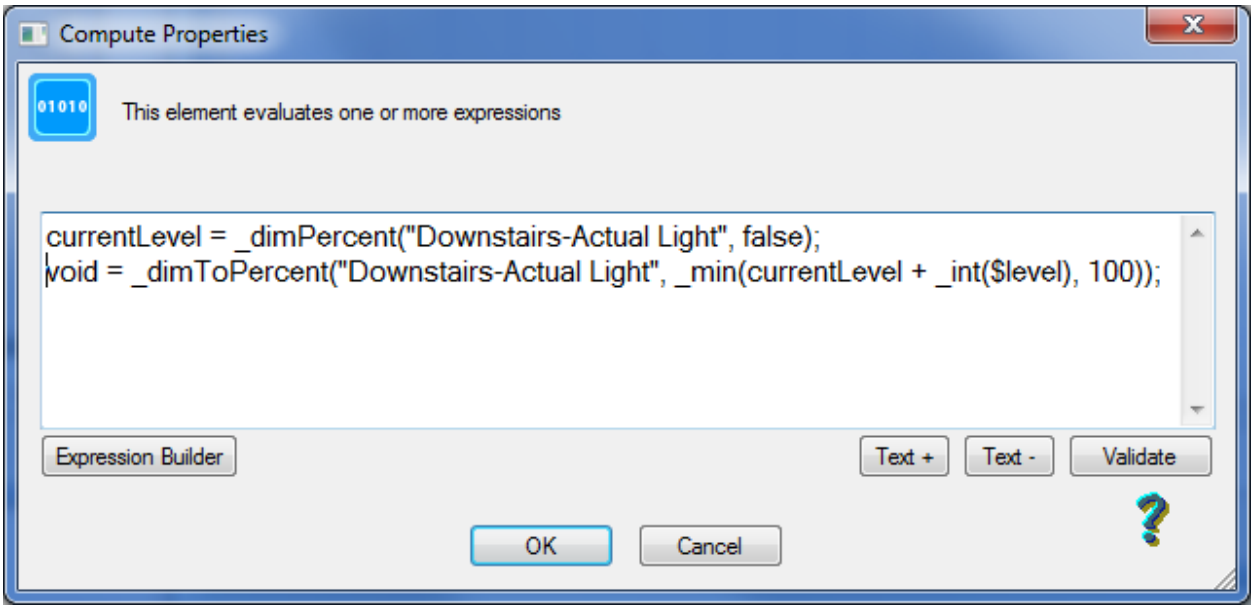
In this example program the first parameter is called "operation" and the second is called "level".

The first two test elements in the program are simply tests to see if it was started by an ON or started by an OFF trigger. If so, then the actual device is controlled ON or OFF.

The next element – a Compute-Test - does this:




This sees if the user told Alexa to increase or brighten the light level. If so, then the Compute element does this:

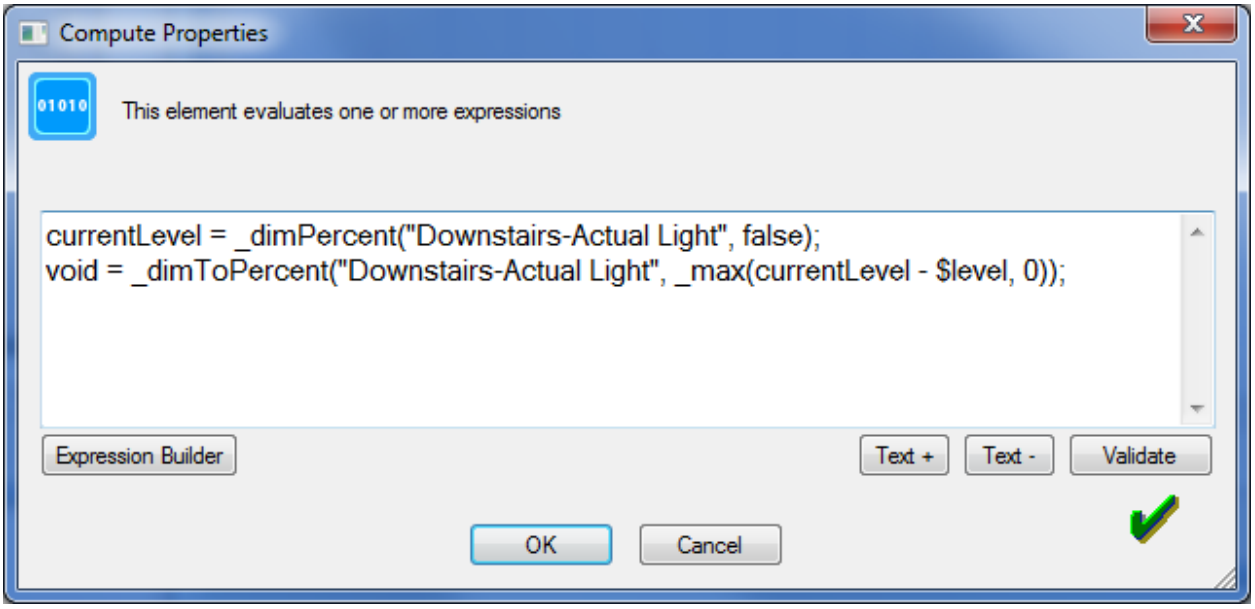



As we saw in an earlier example, the current level of the light can be determined using the _dimPercent function. In this case an optional second argument is provided. If that argument is false it says to not send a message to the device to retrieve its status but instead to use what HCA thinks it is. By not sending a waiting for a response this makes the whole operation faster.

The next statement uses the _dimToPercent function to change the level by taking the current level and adding in the expected change amount. It makes sure that the value doesn't exceed 100.

Note: Why is the _int function involved with the level parameter? That's because the level comes from Alexa as a string and as we learned in the first example the + operator would turn this into a string operation and we want actual addition.

The next Compute-Test element tests to see if the operation is "dimdown" and if so executes a Compute element similar to the "dimup" case.

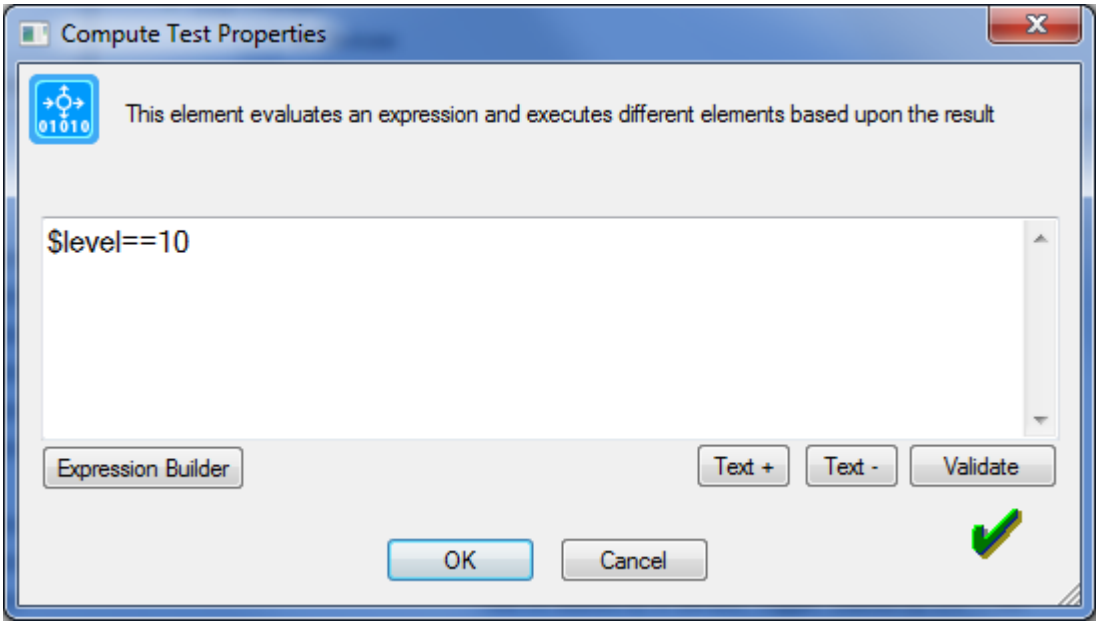



The only difference here is that we are subtracting from the current level the amount of change desired and keeping that value above zero.

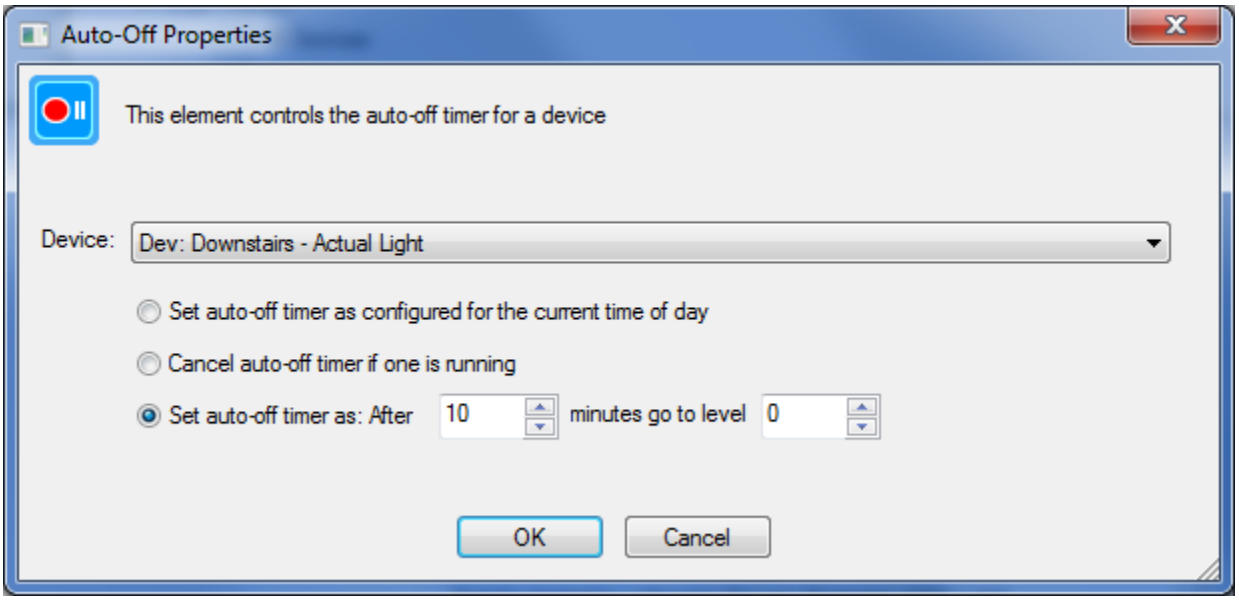
If the operation wasn't ON, OFF, "dimdown" or "dimup" then it must be in response to this said to Alexa and is what we are trying to handle in a special manner.

*Alexa, set downstairs light to <number>*

The first thing to do is to turn the light on and then test the value of the level to see if it is 10, 20, or something else.




And if so, the Auto Off element is executed:




A similar thing is done for the case of 20 and if it wasn't 10 or 20 then the auto off is set to 30.

So when you say "Alexa, set downstairs light to 20" the light comes on and an auto off is set to 20 minutes.

Why not take any possible value from 0 to 100 as the number of minutes? Well, there is no way to get a value into the Auto-Off element nor is there a function for Auto-Off. What a shame! But this is good enough for most uses. If you wanted to example the program to take other auto off minute values that would be simple.

## Final Words

Don't assume that expressions are a way into a secret world in HCA. They have their place but there is no reason to use them when there is an existing program element that does what you want. For example, there is a function that controls a device to ON and is called – not surprisingly - *_On*. But don't use it unless you have a good reason. If you just want to turn on the "Kitchen – Light" use the ON element and not the function.

There are two important factors with expressions that you should always remember.

First, the names of devices in an expression are given in text and are not updated if the object changes its name. For example, if the oft-mentioned "Kitchen – light" becomes "Kitchen – Ceiling Lights" then any ON, OFF, DIM, etc. elements handle that change just fine and don't need to be updated. However, any Compute elements where you are using the device name now become incorrect after the name change and must be found and corrected.

Second, keep in mind is what was covered at the Level Restore example: Variables are global and you should take care that they are not modified by other programs running unexpectedly.

##end##