HCA 201 – Part 2

Last week we looked at a tiled display designed to act as a status display. This week we get into the programs to implement it.

Note: Much of what follows uses the Compute and Compute-Test elements and expressions. Now would be a good idea to review those chapters in the user guide if you are not yet familiar with them.

First let's talk about colors

If you are like me, you spend much time (yes, too much) trying to get colors just right and I change them a lot. What I wanted to do is to have a place where I could define the colors I wanted and not reference them all over the place. In an HCA expression you create a color using the RGB function like this:

```
Color = _RGB(9, 42, 58); // A dark blue
```

A best practice in HCA is to create things like this – what programmers call "magic" numbers - as global variables. The advantage of this comes when you need to change the value. You need only make that change in one place – the program that defines them – and not have to change many other programs where they are used.

In my case I created a program named StatusInit that does this. It has one Compute statement with this in it:

```
ColorOnBack = _RGB(255, 255, 160); // Yellow
ColorOnText = _RGB(0, 0, 0);            // Black

ColorOffBack = _RGB(229, 229, 229); // Light Gray
ColorOffText = _RGB(9, 42, 58);  // Blue

ColorSuspendBack = _RGB(31, 114, 70); // Green
ColorSuspendText = _RGB(0, 0, 0);         // Black

ColorErrorBack = _RGB(254, 21, 81);      // Red
ColorErrorText = _RGB(255, 255, 255);    // White

ColorWorkingBack = _RGB(96, 199, 238);    // Blue
ColorWorkingText = _RGB(255, 255, 255);  // White

ColorHideBack = _RGB(9, 42, 58);
ColorHideText = _RGB(9, 42, 58);
```

These are the colors that I'll use in other programs and the variable name is always used. This way the RGB function that defines the color is only in a single place.

One program or several?

With any task it is almost always a good idea to break it down into smaller sub-tasks. Engineers call this "piecewise decomposition". This is especially needed in HCA as Visual Programs that are too large can lose their "visual" nature and make it hard to see what is going on.

To that end, let's break down the solution into these parts:

- Handling the interface tiles
- Handling the alert tiles
- Handling the room tiles.

And if possible, lets break the last task into two different parts: Generating the list of rooms and handling a single room.

Some people like to work "top down" and others "bottom up". Sometimes it is simplest to start at the top and assume that the lower levels can be implemented – let's first create the program that invokes each task before we create the programs to do the tasks. I prefer in this case to work "bottom up". I'll create the programs for each task and then create a program that invokes each one in turn.

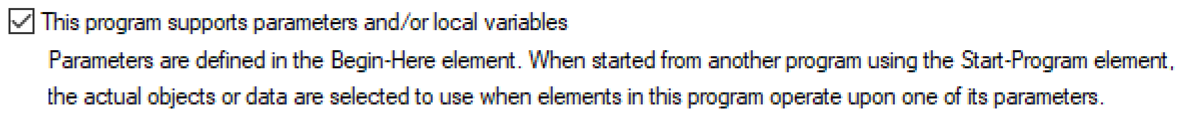
Built for sharing

When creating programs in your own design you can do as you please. But when creating a program to share with others, there are several items you should consider:

- My design and yours may have differently named rooms and a different number of rooms
- My design and yours may have differently named devices
- My design and yours may have different interfaces connected
- My design and yours may have different global variables
- If possible, it is best not to require the user who imports your work to make any changes to their configuration, for example, having to configure the alert manager to run a program. Sometimes this can be unavoidable but is still a goal.

To handle several of these considerations, and no programs refer to a device or room by name, it is necessary to use the DesinOpen and DesignName functions in HCA programs. These functions create lists of elements of the design and lets you operate on them. Later I'll described these more completely but first let's turn to variables.

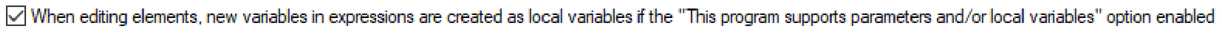
In HCA there are two types of variables: Global and local. A global variable is one that any program can access. All the variables you viewed in the variable inventory dialog are global. A local variable is one that gets created when a program executes and goes away when it terminates. Think of a local variable as a "scratch pad" that a program writes on. Programs, by default don't get local variables, you have to ask for it. After creating a program, open the properties and look on the "Advanced Options" tab. This is the key option:


☑ This program supports parameters and/or local variables

Parameters are defined in the Begin-Here element. When started from another program using the Start-Program element, the actual objects or data are selected to use when elements in this program operate upon one of its parameters.

As the text says, when you tick this box then you can create and use local variables.

Tip: A great option to enable in HCA Options is on the "Visual Programmer" tab:


☑ When editing elements, new variables in expressions are created as local variables if the "This program supports parameters and/or local variables" option enabled

Once you tick this option then any variable that is used in an expression that doesn't already exist as a global variable becomes a local variable. The program can have up to 16 local variables – and they go fast so that's another good reason to keep programs from becoming too large.

The advantage of local variables, especially when sharing programs, is that you will not run into conflicts over global variable use. For example, suppose a design has a global variable called "count" and your program also used a global variable called "count". When your program is imported into that design, the value of the variable "count" could get changed but other programs you were not aware of and break your program.

In all the programs described below, the only global variables are the various colors discussed in the previous section. All other variables are local to the programs.

Room Status

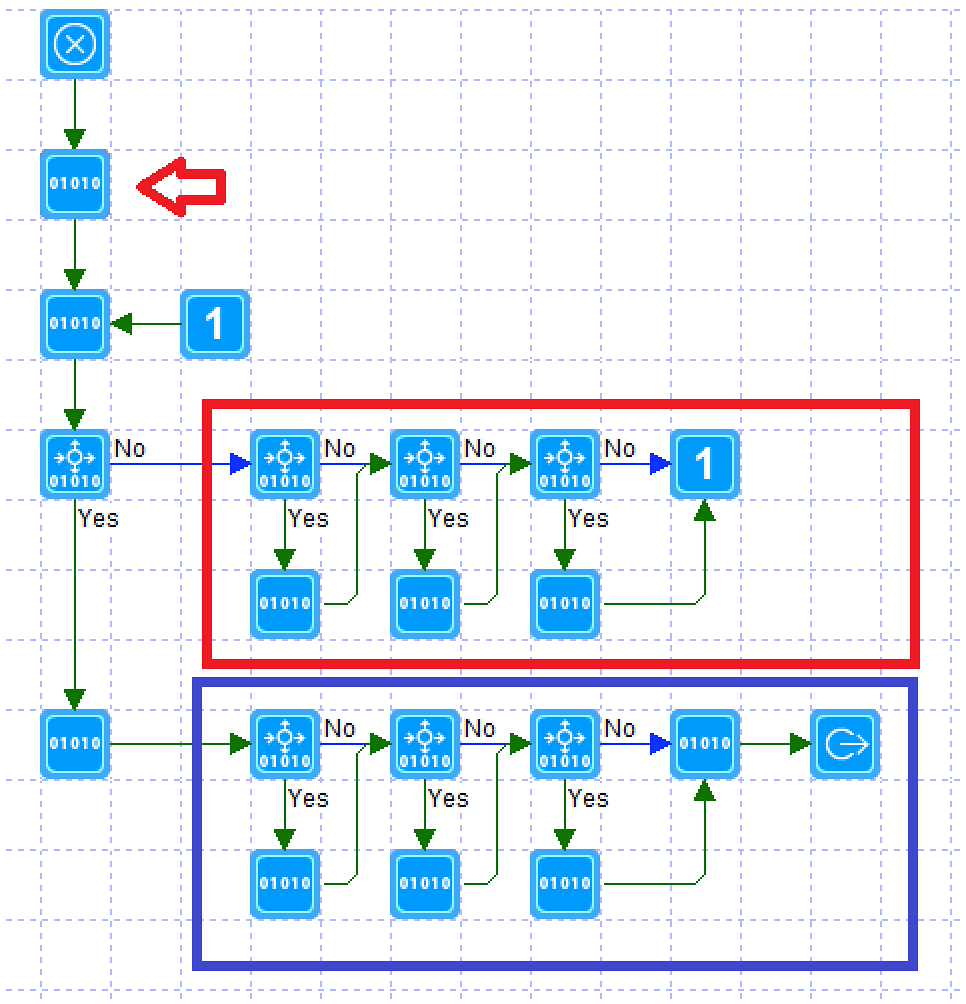
The first program to look at is the one that handles a specific room. It needs to implement these tasks:

1. Find all the devices in the room.
2. For each device determine if it is off, on, suspended, or in an error state and keep track of that
3. Determine the color for the tile: if any device is in an error state, use the error color. If any device is suspended, use the suspended color. If any device ON, use the ON color, otherwise use the OFF color.

The way this is done is to use the DesignOpen and DesignName functions. These are all documented in here <link>.

When using DesignOpen you provide the criterion for locating objects in the design and it returns a number - what programmers call a "handle". Then the DesignName function is used repeatedly. Each time a new name that fits the criterion supplied to DesignOpen is returned. If the name is an empty string, then there are no more objects that fit the criterion. On each use of DesignName, the name is then used with other functions as you need.

Let's look at the program.



The Compute element at the top (red arrow) contains this:

```
hDesign=_DesignOpen(1, $RoomName);
anyDeviceOn = FALSE;
anyDeviceSuspended = FALSE;
anyDeviceError = FALSE;
```

What is happening here is DesignOpen is used to ask for all the devices in the room – that is what the first argument of “1” does. Let’s just hold the question on the odd name with the $ for now – just assume that it is the room name. Also, three variables are initialized to FALSE. Each keeps track of if any device is in those states.

The next Compute element contains this:

```
device = _DesignName(hDesign);
```

This starts the process of getting each device in the room. As described above we need to check that the name is not “” which would be the end of the list. The Compute-Test element handles that.

The red boxed section of the program handles the case where the name is not “” – we have a device to process. The program has three Compute-Test elements using the IsOn, IsSuspended, and IsInErrorState functions. If the test passes, then the variables that keeps track of that state are set to TRUE. When that one device is finished being checked, control goes back to the Compute element where the DesignName function is again used to get the next name.

Note: The IsOn function can have the effect of polling the device for status and you really don’t want to do that! That would take a lot of time to poll all your devices, and just isn’t wanted for this. The IsOn function has an optional second argument which is a yes/no value. If its value is “no”, then device isn’t polled and the status as HCA has it recorded is used. It is details like these that can make or break an implementation.

Best Practice: The Visual programmer tries very hard to draw nice lines between elements but sometimes it just can’t. I use the Connector elements when the line would be ugly. That’s not just a matter of looking good, because a “clean” layout can help with visualization of the program flow.

The section of the program in the blue box looks at the state keeping variables (anyDeviceOn, anyDeviceSuspended, anyDeviceError) and determines the tile background and text color as needed by assigning to local variables. The order of the three tests implement the priority of the colors. The Compute element outside the section in the blue box sets the local variables for color to use to the OFF background and text color. The three tests then determine if that color choice should be overridden. When finally control reaches the last element before the Exit, the colors have been determined.

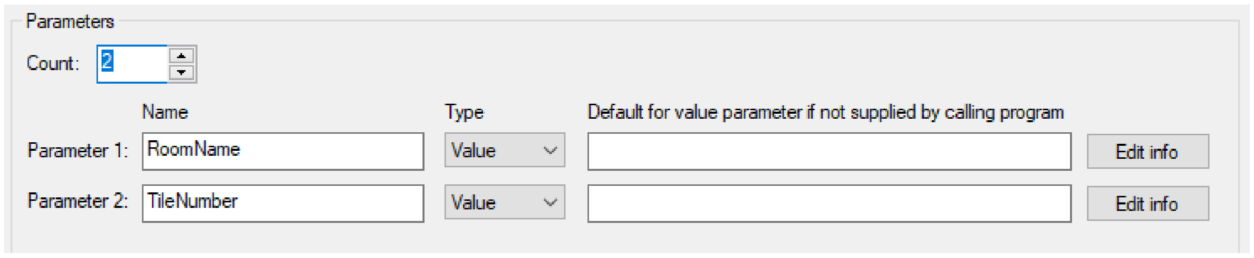
The final Compute element contains this:

```
void=_TileUpdate("Room" + $TileNumber,1,backColor, textColor);
void=_TileUpdate("Room" + $TileNumber,0,$RoomName);
```

The TileUpdate function is the key to changing the tile colors. It takes the name of the tile and a code to determine what it does. The first TileUpdate sets the color and the second changes the tile label. The tile name is formed from “Room” suffixed by a number. In last week’s installment when the tiles were created, each of the room tiles was named “Room” followed by a number. All we need do is to take the tile number and the name of the tile is easily constructed.

OK, I’ve avoided long enough: Where does the room name and tile number come from? From program parameters.

There is documentation on this topic that you should review. Briefly, the idea is that when a program is started it can be supplied from one to eight pieces of information. The program can then process those pieces of information in a similar way as it handles local variables. The names these “pieces of information” (called “parameters” or “arguments” in programmer speak) are given in the Begin-Here element’s properties.



Parameters

Count: 2

	Name	Type	Default for value parameter if not supplied by calling program	
Parameter 1:	RoomName	Value		Edit info
Parameter 2:	TileNumber	Value		Edit info

This program uses two: The name of the room and the number of the tile used for that room. In expressions, a parameter is used with the name prefixed by a $. In this program when we used $RoomName in an expression, I was using whatever room name was supplied when the program started.

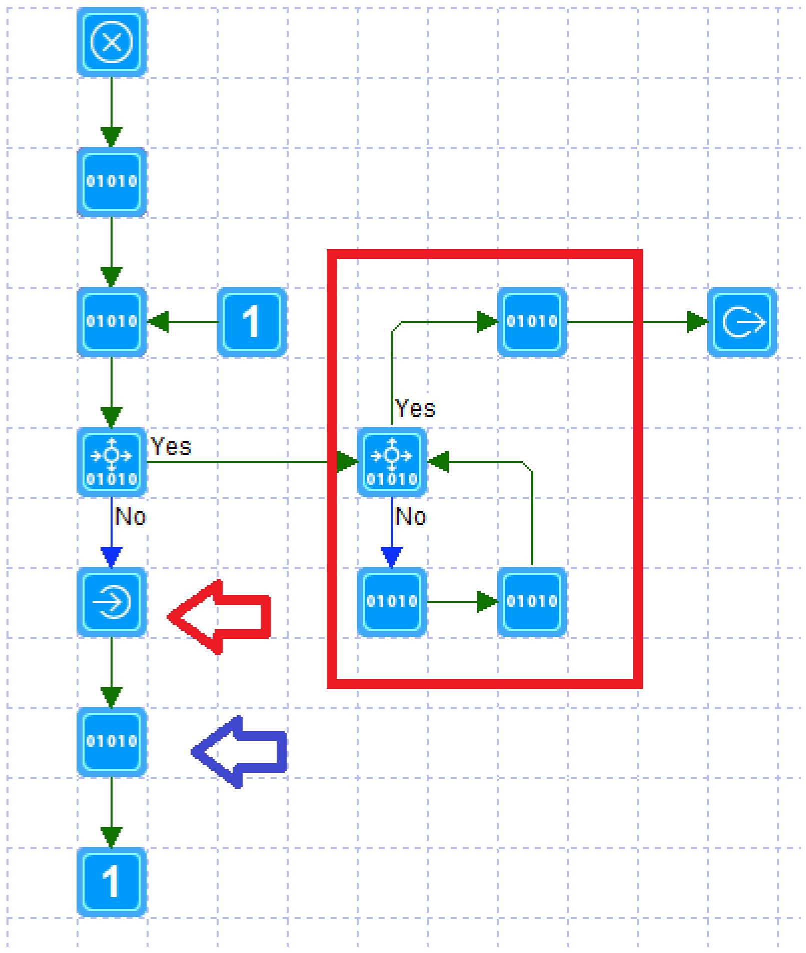
The major advantage of a program with parameters is that it can perform whatever action it does on different items. If this program is supplied with the room name of "Kitchen" it will look at all the devices in the Kitchen. If it is provided with the room name of "Library" it looks at all the devices in the Library. And so on. Instead of needing a separate program for each room we need only a single program.

All Rooms Status

Since we now have a program that handles the status for a single room, let's look at the program that handles all rooms. Its tasks are:

1. Create a list of all rooms.
2. Use the room status program on each room up to 16 rooms
3. The tiles for any unused rooms should be hidden

This program, like the last one, uses the same DesignOpen and DesignName functions to generate a list of rooms and then starts the RoomStatus program on each. Here it is:



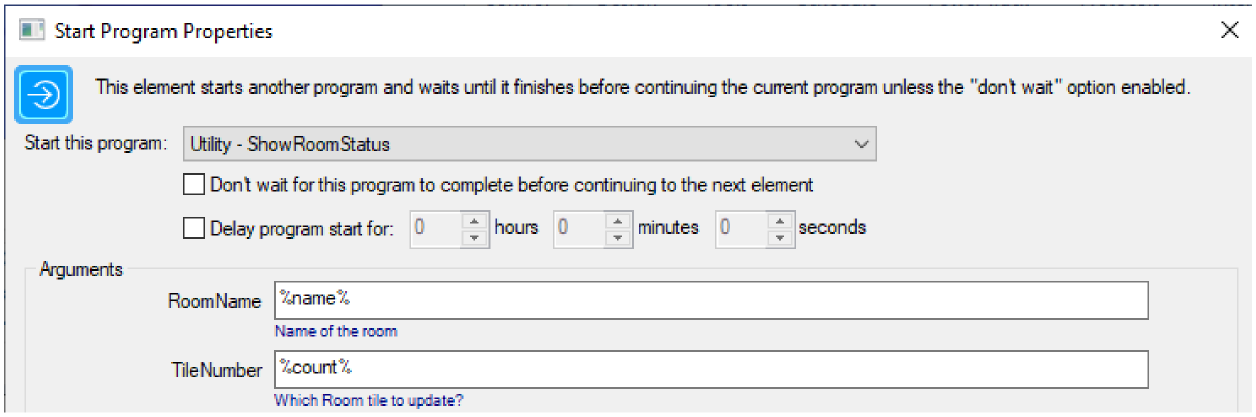
The first Compute element uses DesignOpen and initializes two local variables.

```
hDesign = _DesignOpen(6);
count=1;
roomMap="";
```

The “6” to DesignOpen tells it to produce a list of rooms. The local variable “count” is used to keep track of which tile is being used as we go through the list of rooms. Ignore “roomMap” for now.

The next Compute element uses the same DesignName function as in the single room status program described above. And it is followed by a test to see if the name is “” which, as you know, tells us we are done with the list.

All the interesting stuff happens in the Start-Program element (the red arrow).


Start Program Properties

This element starts another program and waits until it finishes before continuing the current program unless the "don't wait" option enabled.

Start this program: Utility - ShowRoomStatus

☐ Don't wait for this program to complete before continuing to the next element

☐ Delay program start for: 0 hours 0 minutes 0 seconds

Arguments

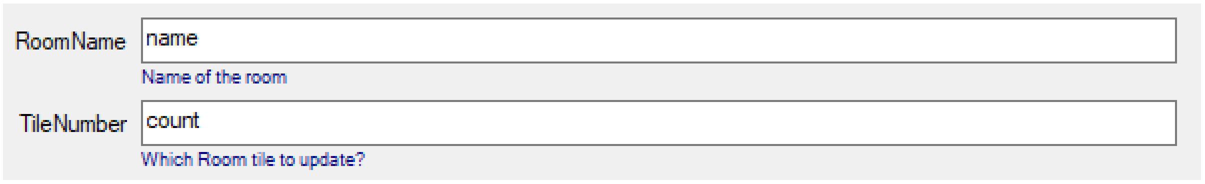
RoomName: %name%
Name of the room

TileNumber: %count%
Which Room tile to update?

First off, the program being started is the individual room status program. It is supplied with two arguments: the name of the room and the tile to update.

The local variables "name" and "count" contain those. We need to supply the values those variables contain to the room status program.

What you type into the argument boxes to a program are the actual strings that the program gets. If we entered it like this:


RoomName: name
Name of the room

TileNumber: count
Which Room tile to update?

The room status program would get the strings "name" and "count" and that's not at all what we want! Like many other places in HCA where you supply a piece of text to an program element, you can embed expressions in that string that get evaluated and the final string produced. To do that you place the expression in %'s and that is what I did. Look at the image of the Start-Program element above.

Next, at the blue arrow, the Compute element contains this:

```
roomMap= roomMap + name + ",";
count=count+1
```

The "count" variable, which is the tile number, is incremented. In the roomMap variable I am building a list of the room names in the order that the tiles are used. This will be used next week when we make this display have a bit more function, so for now just ignore it.

The careful reader may be wondering what happens if there are more than 16 rooms? This program doesn't check for that – it could but I elected not to. What happens is that the single room status program creates tile names like Room17, Room18, etc that don't exist. The TileUpdate element doesn't care. It just doesn't update any tile. Is this taking advantage of a "nit" in HCA? Maybe but it works.

The final section of the program contains the elements in the red box. What that does there is to hide any tiles not used. It does this by hiding tile numbered "count" to 16. Each time a tile was used, the count variable got incremented so the value in "count" after processing all rooms is the number of the first unused tile. All it needs to do is to hide the “count” tile, increment count, then hide that tile, increment count, etc all the way until count is greater than 16.

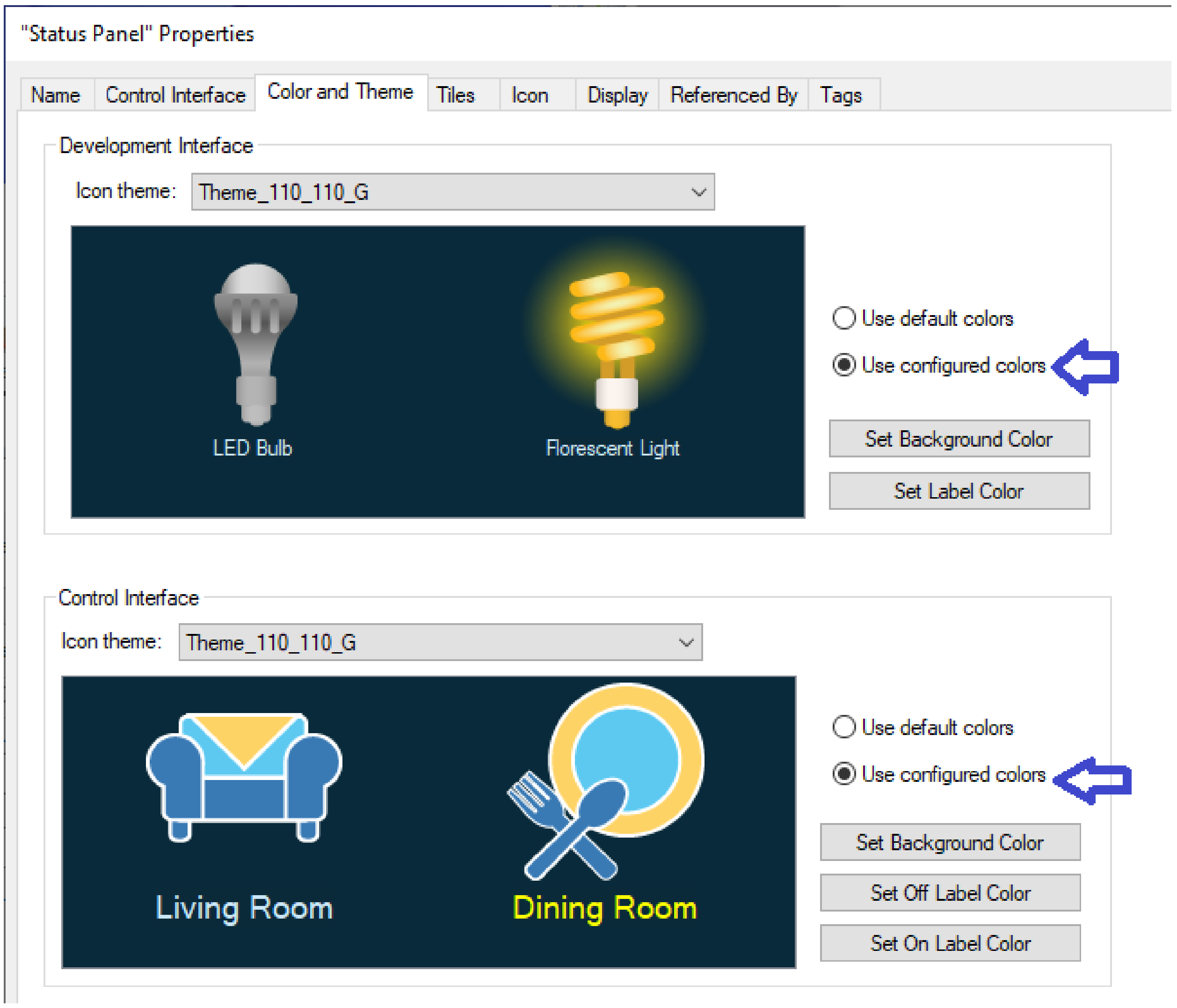
How is a tile hidden? Its background color and text color are set to the same color that the whole display has – it’s still there it just can’t be seen. And that’s clever, right? Yes, but that’s a problem since we can’t know what that color is! In doing most tasks you have to sometimes “work around” problems. When cooking sometimes the recipe calls for an ingredient that you don’t have so you must substitute. So too in this case. What we want to know is what the background color of the display is. There are three difficulties

1. The user who imports this package could have set a different default background color.
2. When the control UI shows this display it probably has a different color than when shown in the development user interface
3. A client could have a different color scheme as well

And there is no complete way to solve this. There is no method in HCA for a program to learn the background color of a display. And, worse, one client could have a different color scheme than another and since all programs are executed on the HCA server, it can’t know the colors those clients use.

What to do? The best I can do is to fix a background color for the development and control user interfaces and make them the same color. This forces any clients to use those colors as well. What about users with other color schemes? Well, they will just have to adjust to what colors I picked, or they can change the program that creates the colors and use a different RGB value.

To set a specific background color for a display and to not use the default, open the display properties and choose the “Color and Theme” tab:


"Status Panel" Properties

Name | Control Interface | Color and Theme | Tiles | Icon | Display | Referenced By | Tags

Development Interface

Icon theme: Theme_110_110_G

( ) Use default colors
(●) Use configured colors

Control Interface

Icon theme: Theme_110_110_G

( ) Use default colors
(●) Use configured colors

So, yes, a “work around” but best we can do.

Interface Status Program

The ShowInterfaceStatus program isn’t much different than you have already seen. Unlike the room programs, HCA can only have up to 8 interfaces so it is a simple matter to just go though each and check its status and get its name. And, like the room programs, the tiles for unused programs can be hidden in the same way.

Alert Status Program

The ShowAlertStatus program is the simpelest of all the programs in this package. All that it contains are only two elements: A begin-Here and a Compute element. The Compute element contains this:

```
countOverdueDevices =_AlertCount(16)+_AlertCount(17)+_AlertCount(18)+_AlertCount(19);
```

```
countProgramErrors = _AlertCount(15);
countUnknownReceptions = _AlertCount(1);
countDesignAlerts = _AlertCount(21)+_AlertCount(22)+_AlertCount(23)+_AlertCount(24);

void = _TileUpdate("Alert_Devices", 1, _IIF(countOverdueDevices==0, ColorOffBack, ColorErrorBack), ColorOffText);

void = _TileUpdate("Alert_Programs", 1, _IIF(countProgramErrors ==0, ColorOffBack, ColorErrorBack), ColorOffText);

void = _TileUpdate("Alert_Receptions", 1, _IIF(countUnknownReceptions ==0, ColorOffBack, ColorErrorBack), ColorOffText);

void = _TileUpdate("Alert_User", 1, _IIF(countDesignAlerts ==0, ColorOffBack, ColorErrorBack), ColorOffText);
```

It uses the TileUpdate element and the color global variables as we have seen many times before plus two new functions: AlertCount and IIF.

AlertCount does what you would expect: returns the count for an alert. The numbers supplied as arguments choose which alert to get the count for. Unfortunately, I must admit that it isn't documented anyplace. (Yes, it is my job and I will update the expression document with it). For now, just know that I have the correct values.

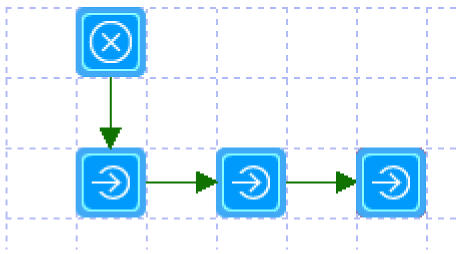
The IIF function is a nice one to know and looks like this:

IIF (expression, value1, value2)

If the expression evaluates to a YES, then the result is value1 otherwise value2. Its in easy way to get some conditional logic in without needing to use a Compute-Test element.

Putting it all together

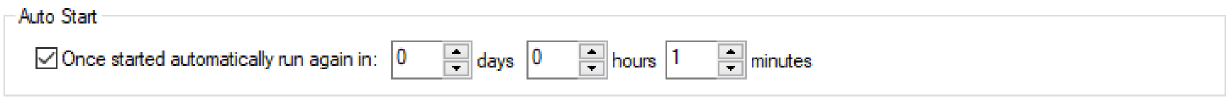
Now that we have all the programs we need, the next step is to create one program to rule them all. I called that program UpdateStatusDisplay and looks like this:



All the program does is to use three start-program elements for the three programs we created: ShowAllRoomsStatus, ShowInterfaceStatus, ShowAlertStatus.

Now there is a decision to make: How often does the display get updated? It would be nice if it got updated when anything changed: A device changed state, an interface connected or disconnected, an alert was generated. The problem is that, in general, there is no way to do that. You could create state change triggers for every device which is not practical, and for each alert configure a program to run when it changes, which is also not practical. And there is no method in HCA to start a program when an interface connects or disconnects.

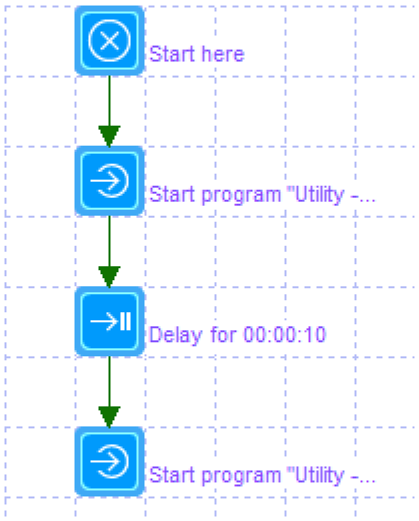
The best we can do is to run the program "every so often". How often is the question. Since the program doesn't do anything but examine state – it doesn't communicate with any devices – it doesn't use many resources. I elected to run the program every minute. The way I did that was to use this feature available on the program's Advanced Options.


Auto Start

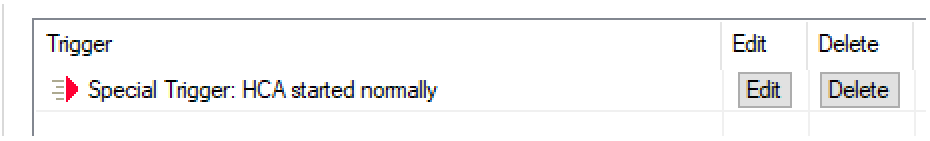
☑ Once started automatically run again in: 0 days 0 hours 1 minutes

What this does is that each time the program finishes, it gets restarted after that countdown expires – in this case 1 minute.

But how does the program get started in the first place? For that I created yet another program that looks like this:



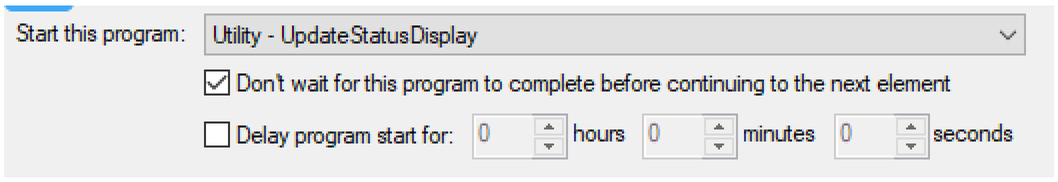
What it does is to start the first program created – the program that sets all the global variables for the colors, then waits 10 seconds then runs the ShowAllRoomsStatus program. Why do this? Because we can configure that program with this trigger:



Trigger	Edit	Delete
Special Trigger: HCA started normally	Edit	Delete

This starts the program when HCA loads the design. It starts the program to create all the colors, then waits 10 seconds – this is to allow all the interfaces to be up to speed before their status is checked – and then starts the ShowAllRoomsStatus.

Here is that start-program element that stats the AllRoomsStatus program. There is one very important feature used: The Don't Wait option.

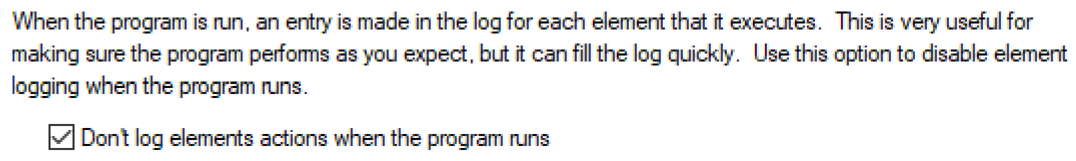

Start this program: Utility - UpdateStatusDisplay

☑ Don't wait for this program to complete before continuing to the next element

☐ Delay program start for: 0 hours 0 minutes 0 seconds

In HCA there are two ways to start a program from another program: A program can run as if all the elements in it were part of the program containing the Start-Program. In programmer speak this is called a “subroutine”. Or it can run as a totally separate program. Because of the way HCA implements programs running as “subroutines” the “auto start” feature described above doesn’t work. So, the program must be started as a sperate program and that is what ticking that option does. Why is it this way? It just is the way it is and like life, some things can’t be changed only gotten used to.

After I tested, and before sharing, I made sure to disable logging for all these programs. Especially programs like these that execute hundreds of elements when they are executed. That can quickly make the log unusable. On the programs properties on the Log tab is this option:


When the program is run, an entry is made in the log for each element that it executes. This is very useful for making sure the program performs as you expect, but it can fill the log quickly. Use this option to disable element logging when the program runs.

☑ Don't log elements actions when the program runs

Making it work

In creating this package, I ran into several problems because I got some aspect of a program wrong. That’s just how it goes. Even though I know HCA well, I still make mistakes. That’s where the program debugger comes in very handy. This is important when working with programs with local variables as you can examine their values only in the debugger since they only exist while the program executes. Getting a working knowledge of the debugger would really help you. Again, there is good documentation on it. Worth a read.

And a last tip, one that can save you much grief: Make sure you save your design often! While developing one of the programs I made the mistake of creating a condition where the program never finished and because I had the log enabled it was generating TONS of log entries so quickly that I couldn’t stop it and had to terminate HCA. But because I had saved the program before I tested, I didn’t lose anything.

Next week, we will see if we can improve this package in a few ways and in doing so encounter some interesting issues to resolve.

##end##