GEORGE HOME AUTOMATION

Developer Guide V1.2

Table of Contents

Introduction

George Home Automation (or GHA) is an application platform designed to support complex home automation scenarios. Its object-oriented design gives end-users the flexibility to create objects that model the everyday items we see in our homes and map those items to underlying devices to control our home environment.

One of the key features of GHA is extensibility. New functionality can be created for GHA using a variety of methods including: Modules, Device drivers, and in-application scripting. In this document, we will discuss each of these capabilities and how end-users can use programming skills to create new functionality for GHA.

GHA is written in C# and leverages Microsoft's .NET development environment.

Intended Audience

The reader is expected to understand C# and Microsoft .Net development. Depending on the type of extensibility or customization the reader is attempting, knowledge of networking and other integration techniques may be required. A working knowledge of object-oriented programming is also required.

Key Concepts

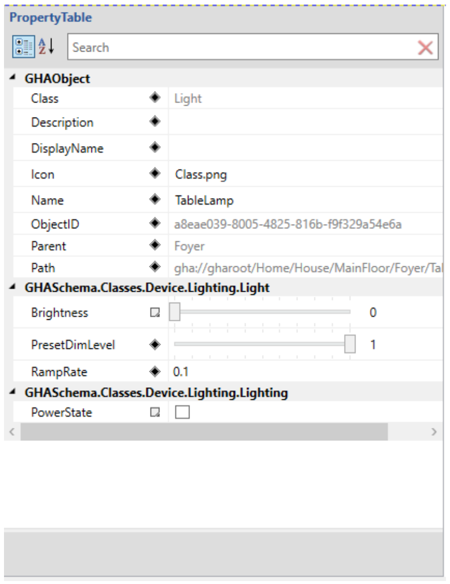
Many of the topics in this section can also be found in the GHA Users Guide. Knowledge of the concepts here are required to develop new functionality for GHA.

There are several concepts that are necessary to understand when starting out with GHA. These are generally described in this section.

GHA Hierarchy

The entire GHA system lives in a hierarchy that contains instances (called GHA Objects) of predefined or custom developed classes used to implement various automation scenarios. The out-of-the-box hierarchy contains the following root nodes:

- Home – Contains models for locations and devices that are part of the automation environment
- Devices – Drivers that implement standard **Capabilities** and control physical devices live here
- Media – Catalog of media managed by GHA
- Scenes – Predetermined collections of device states that can be activated and deactivated manually or through automation
- Modules – Provides for the creation of custom classes to implement new GHA capabilities
- Information Sources – Contains services that provide data from external sources (e.g. Weather)
- Monitors – Provides various monitors for GHA subsystems (i.e. Alerts, Task Scheduler)
- Server – Contains configuration nodes for various server functions



PropertyTable

Search

GHAObject	
Class	Light
Description	
DisplayName	
Icon	Class.png
Name	TableLamp
ObjectID	a8eae039-8005-4825-816b-f9f329a54e6a
Parent	Foyer
Path	gha://gharoot/Home/House/MainFloor/Foyer/Ta
GHASchema.Classes.Device.Lighting.Light	
Brightness	0
PresetDimLevel	1
RampRate	0.1
GHASchema.Classes.Device.Lighting.Lighting	
PowerState	☐

Figure 1 GHA Object with Properties

GHA Objects

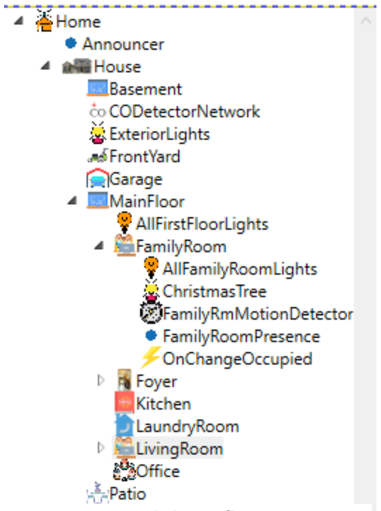
GHA Objects in the hierarchy are *instances* of classes that implement various functions. Each GHA Object has a unique Object ID that is represented as a Globally Unique Identifier (GUID). Each object can be references by its GUID or a *Path*. The Path is also unique and represents the object's location in the hierarchy. The path is a universal resource identifier (URI) starting with the prefix "gha://". Succeeding parts of the path describe the lineage of the object. For example, an object with the path: gha://gharoot/Home/House/MainFloor/Foyer/Chandelier shows where in the hierarchy the Chandelier object resides. The Chandelier's parent is an object called "Foyer". The Foyer's parent is an object called "MainFloor", and so-on. A GHA Object may have at most *one* parent and have many children.

GHA Objects contain various properties that describe the object itself and its current state. All GHA Objects contain the default properties Class, Description, DisplayName, Icon, Name, ObjectID, Parent, and Path. Object-specific properties are also defined for objects to describe its state. For example, the object shown in Figure (1) represents a Table Lamp which is an instance of the *Light* class.

In addition to the standard properties associated with every GHA object, the *Light* class shown here has properties unique to it: Brightness, PresetDimLevel, RampRate, and PowerState. It is also worth noting that this object is a good example of inheritance. As an object-oriented platform GHA makes heavy use of inheritance to simplify the creation of new capabilities. We will talk more about inheritance when we discuss custom modules.

The Home node

The Home node in GHA is used to model the space to be automated. Locations may be specified under the Home node. For example, a *House* object is normally the first object placed under the Home node. While not strictly a requirement, it is a best-practice. The House object can then contain various objects used to provide logical separation (i.e. floors, garage, basement, etc.). These objects can contain children to further subdivide the space (i.e. bedrooms, offices, etc.). Finally, device objects used to represent physical devices can be placed anywhere under the Home node hierarchy. For example, a Family Room object can contain devices model objects that represent lights, motion detectors, presence detectors, thermostats, etc. Figure (2) shows an example of a Home node and its hierarchy containing a typical three-floor home.



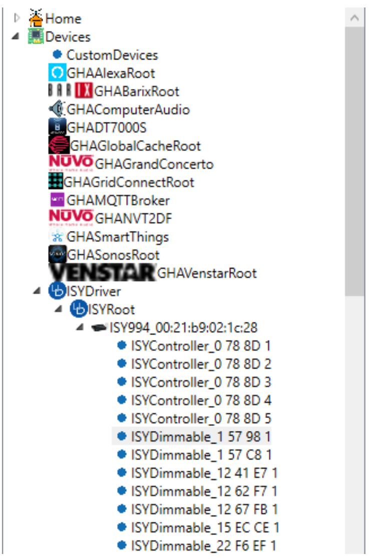
```
Home
  Announcer
  House
    Basement
    CODetectorNetwork
    ExteriorLights
    FrontYard
    Garage
    MainFloor
      AllFirstFloorLights
      FamilyRoom
        AllFamilyRoomLights
        ChristmasTree
        FamilyRmMotionDetector
        FamilyRoomPresence
        OnChangeOccupied
      Foyer
      Kitchen
      LaundryRoom
      LivingRoom
      Office
    Patio
```

Figure 2 Typical three-floor Home hierarchy

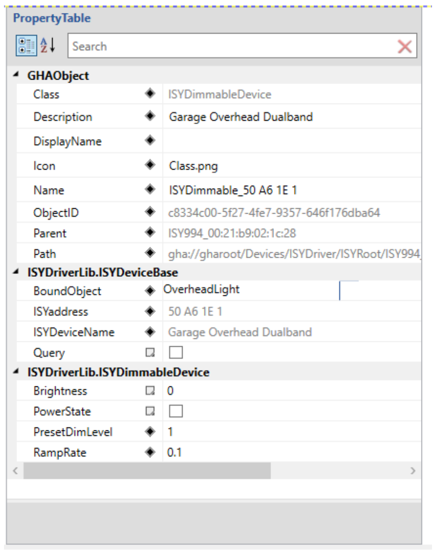
The Home node hierarchy in GHA is special in that it separates the physical device controllers (i.e. Insteon, SmartThings, etc.) from logical representations of devices. Logical representations "live" under the Home node in the GHA hierarchy. These logical representations are objects that contain certain **Capabilities**. When used in a Home level object, **Capabilities** define the user-level specification of a device. For example, a Light contains the **Capabilities** *PowerState* and *Dimmable*. Those **Capabilities** manifest themselves as properties in the Home level object. In Figure (1), the PowerState property represents the *PowerState* capability as do the Brightness, PresetDimLevel, and RampRate properties represent the *Dimmable* capability. The physical device controllers themselves are represented under the Devices node.

The Devices Node

GHA has a standard Application Programming Interface (API) which is used implement device drivers. The drivers delivered with GHA or those developed by third parties, leverage this API and **Capabilities** to implement a consistent control mechanism between Home level devices and physical devices. The Devices node contains the drivers configured for a GHA system. The drivers themselves are represented as GHA Objects under the Devices node. And, like all GHA Object, device driver nodes can also contain children. Figure (3) shows object hierarchy associated with the Universal Devices ISY driver including Its child nodes. Figure (4) shows the properties associated with one of the children of the ISY device. Consistent with its name, this object implements control over a dimmer. You can see in the property list, the standard GHA Object properties as well as properties the implement the *PowerState* and *Dimmable* **Capabilities.** These are the same **Capabilities** that were used to define the properties for the Home level device. In this case, however, the properties that implement the desired **Capabilities** have code associated with them to command the ISY to set the state of the physical dimmer.



```
Home
Devices
    CustomDevices
    GHAAlexaRoot
    GHABarixRoot
    GHAComputerAudio
    GHADT7000S
    GHAGlobalCacheRoot
    GHAGrandConcerto
    GHAGridConnectRoot
    GHAMQTTBroker
    GHANVT2DF
    GHASmartThings
    GHASonosRoot
    GHAVenstarRoot
  ISYDriver
    ISYRoot
      ISY994_00:21:b9:02:1c:28
        ISYController_0 78 8D 1
        ISYController_0 78 8D 2
        ISYController_0 78 8D 3
        ISYController_0 78 8D 4
        ISYController_0 78 8D 5
        ISYDimmable_1 57 98 1
        ISYDimmable_1 57 C8 1
        ISYDimmable_12 41 E7 1
        ISYDimmable_12 62 F7 1
        ISYDimmable_12 67 FB 1
        ISYDimmable_15 EC CE 1
        ISYDimmable_22 F6 EF 1
```



PropertyTable

Search

GHAObject	
Class	ISYDimmableDevice
Description	Garage Overhead Dualband
DisplayName	
Icon	Class.png
Name	ISYDimmable_50 A6 1E 1
ObjectID	c8334c00-5f27-4fe7-9357-646f176dba64
Parent	ISY994_00:21:b9:02:1c:28
Path	gha://gharoot/Devices/ISYDriver/ISYRoot/ISY994
ISYDriverLib.ISYDeviceBase	
BoundObject	OverheadLight
ISYaddress	50 A6 1E 1
ISYDeviceName	Garage Overhead Dualband
Query	☐
ISYDriverLib.ISYDimmableDevice	
Brightness	0
PowerState	☐
PresetDimLevel	1
RampRate	0.1

Figure 3 ISY Object properties

The Binding

A key concept to understand in GHA is that logical devices in the Home hierarchy are connected to device objects under the Devices hierarchy. We have this level of abstraction so that multiple physical device technologies can be consistently mapped to Home hierarchy objects without the user worrying about the differences between say Insteon and SmartThings. Another benefit of this approach is to allow for the replacing of underlying device technologies without impacting any of the objects under the Home hierarchy and associated automations. The connection between the logical representation of devices under the Home hierarchy and the device driver representation under the Devices node is called a **Binding**. During a **Binding** operation, logical devices search for physical device controllers with matching **Capabilities**. The user selects the appropriate device, and the two objects are bound together. A property changed in one, will be reflected in the other.

Scenes

A Scene in GHA is a mechanism where multiple objects can be controlled collectively. Let's say you want to define a Scene called "ChristmasDecorations" where you want to control yard lights connected to your Landscape lighting and your Christmas tree together. In GHA, this is simply done by creating a Scene object under the Scenes node, adding the objects to control, setting the desired property values for those objects and deciding on how you want the scene to behave. The key concept to understand about Scenes is the Scene behavior. In GHA, scenes can be configured to set the desired object properties when activated and leave them that way even when the scene is deactivated. This is called Set behavior. Alternatively, the scene can be configured to set the desired properties when the scene is activated and restore them to their previous state when deactivated. This is called Set-Restore behavior.

Schedules

Schedules in GHA can be used to activate a Scene or execute a script at a given date and time. The actions can be taken just once or repeated. In addition to simple date/time triggers, GHA can schedule actions for local sunrise and/or sunset.

Customizations

GHA can be customized in several ways. First, all objects in the Home part of the hierarchy can have a custom script attached to a change of any of its property values. For example, all locations implement the *Occupancy* **Capability** which includes a property called Occupied. When Occupied is true, the location is considered occupied. When false, the location is considered empty. A property change script can be associated with the Occupied property and it will be invoked when the Occupied property changes. The script can then perform custom actions based on the occupancy of the location.

The second method for customization is the Module. There is a Modules node in the hierarchy which contains user-developed modules that can be used to implement highly sophisticated customizations. The modules themselves contain custom classes which follow the same object-oriented methodology that built-in classes follow. As such, they can contain properties and methods as well as support inheritance. Once a module has been created, it is compiled and made available to the rest of GHA. Classes within modules implicitly conform to the GHA Object specification and can be used anywhere a built-in GHA Object can be used.

The final way to do customizations is through building custom device drivers. GHA exposes an open API that can be leveraged by developers to implement their own device drivers.

Decorations

GHA uses .NET Decorations to control the handling of various properties and classes. These are described here.

GHAProperty

	Name	Description
	GHAPropertyAttribute	Adds specific attributes to GHA Properties for handling by consuming objects

Properties

	Name	Description
	Aggregation	Defines the method used by parent objects to aggregate the property from children
	Description	Text stating the purpose of the property
	IsBindable	Property may be used as a binding point
	IsEnumeration	Tells GHA that this property is an enumeration
	IsMomentary	Applies to boolean values. Property resets to false automatically.
	IsReadonly	Property should not be changed by GUI's
	IsVisible	Determines if the property is visible to UI's
	SpecialHandling	An application-specific field that can be used tag the property for special handling (i.e. GUI rendering a specific control to display the property)

GHA Executive Services available to developers

GHA uses a component called the Executive to coordinate and manage the activities of the entire system. The Executive offers several services that make it easier for developers to perform routine, repeatable tasks. They are exposed to developers as static methods in the GHASys class. They are summarized below.

	Name	Description
S	AddTimer	Creates a simple timer object under Modules/GHATimers node in the hierarchy. Timer is NOT started after creation. Set the Enabled property to true to activate the timer
S	CreateAlert(DateTime, String, String, String, String)	Creates and publishes an alert
S	FireDebugCallbacks(String, String, String, String, Int32, String)	Initiates Debug event callbacks

	FireMessageCallback(String, String)	Initiates a message callback. Receiving end should pop a message box or take some sort of action to let the user know this event has happened
	GetDevices	Returns the Devices root object in the GHA hierarchy
	GetHome	Returns the Home object in the GHA hierarchy
	GetMedia	Returns the Media root object in the GHA hierarchy
	GetModules	Returns the Module root object in the GHA hierarchy
	GetMonitoring	Returns the Monitor root object in the GHA hierarchy
	GetScenes	Returns the Scenes root object in the GHA hierarchy
	GetSchedules	Returns the Schedule root object in the GHA hierarchy
	IsDark	Helper method to access Nighttime status
	IsLight	Helper method to access Daylight status
	RemoveTimer	Removes the timer with the specified name
	SendCommand(IGHAObject, string)	Sends the provided string using the command device. The command device is assumed to implement either the IInfraredOutput or ISerialPort capabilities
	SendSMTPMessage(String, String, String, String, String)	Queues an SMTP message
	SendSMTPMessage(String, String, String, String, String, String, String)	Queues an SMTP message
	Speak(String)	Initiate text to speech operation that will render on the default audio device of the GHA server
	Speak(String, ITTSStream)	Initiates text to speech operation that will render on the specified device
	Speak(String, ITTSText)	Initiates text to speech operation that will render on the specified device

The IGHAObject (a deeper dive)

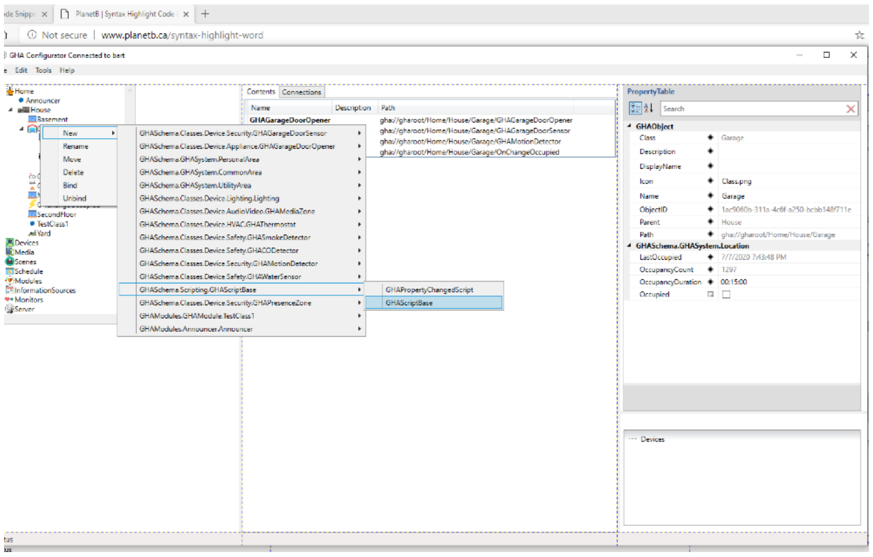
In this section we will learn a little more about the IGHAObject. We will build a couple of simple scripts exercising some of the methods found in the IGHAObject. Finally, we will talk a little about GHAHelper functions.

In addition to the properties described earlier, the IGHAObject contains several methods which are used to manage itself and the hierarchy. A complete definition of the IGHAObject can be found in appendix I. However, there are a few methods which are the most used. These are shown in Table I.

Table 1Most used IGHAObject methds

	Name	Description
	CreateObject	Creates an object of the specified class name as a child of the calling object
	DeleteObject	Deletes a child object of the calling object matching the specified name
	GetObject(Guid)	Returns the object with the requested Object ID
	GetObject(String)	Returns the object at the specified relative path
	GetObject(Uri)	Returns an object at the specified absolute Uri
	GetValue	Returns the value of the specified property
	Move	Moves the current object and its children to a new parent object
	Rename	Renames the calling object to the specified new name
	SetValue(String, Object)	Sets the specified property to the specified value
	SubscribeToCreate	Subscribes to the Create object event for the object. Invokes event handler if a new object is created under the owning object.
	SubscribeToDelete	Subscribes to the Delete object event for the object. Invokes event handler if a child object is deleted under the owning object.
	SubscribeToProperty	Subscribes to any changes to the specified property
	Unsubscribe	Unsubscribes from the specified subscription (property change, create, or delete)

Whether called from a script, Module, or device driver; these methods allow for the creation, manipulation, and deletion of objects within the GHA hierarchy. Let us start with a couple of simple examples of how these methods are used in a script.


de Snipp
PlanetB | Syntax Highlight Code
Not secure | www.planetb.ca/syntax-highlight-word
GHA Configurator Connected to bart
Edit
Tools
Help
Home
Announcer
House
Basement
SecondFloor
TestClass1
Yard
Devices
Media
Scenes
Schedule
Modules
InformationSources
Monitors
Server
New
Rename
Move
Delete
Bind
Unbind
GHASchema.Classes.Device.Security.GHAGarageDoorSensor
GHASchema.Classes.Device.Appliance.GHAGarageDoorOpener
GHASchema.GHASystem.PersonalArea
GHASchema.GHASystem.CommonArea
GHASchema.GHASystem.UtilityArea
GHASchema.Classes.Device.Lighting.Lighting
GHASchema.Classes.Device.AudioVideo.GHAMediaZone
GHASchema.Classes.Device.HVAC.GHAThermostat
GHASchema.Classes.Device.Safety.GHASmokeDetector
GHASchema.Classes.Device.Safety.GHACODetector
GHASchema.Classes.Device.Security.GHAMotionDetector
GHASchema.Classes.Device.Safety.GHAWaterSensor
GHASchema.Scripting.GHAScriptBase
GHASchema.Classes.Device.Security.GHAPresenceZone
GHAModules.GHAModule.TestClass1
GHAModules.Announcer.Announcer
GHAPropertyChangedScript
GHAScriptBase
Contents
Connections
PropertyTable
Search
Devices

Name	Description	Path
GHAGarageDoorOpener		gha://gharoot/Home/House/Garage/GHAGarageDoorOpener
		gha://gharoot/Home/House/Garage/GHAGarageDoorSensor
		gha://gharoot/Home/House/Garage/GHAMotionDetector
		gha://gharoot/Home/House/Garage/OnChangeOccupied

GHAObject	
Class	Garage
Description	
DisplayName	
Icon	Class.png
Name	Garage
ObjectID	1ac9060b-311a-4c6f-a250-bcbb148f711e
Parent	House
Path	gha://gharoot/Home/House/Garage
GHASchema.GHASystem.Location	
LastOccupied	7/7/2020 7:43:48 PM
OccupancyCount	1297
OccupancyDuration	00:15:00
Occupied	☐

Our first example will be a script that is invoked when the Occupied property changes for a location object called Garage. You start by right clicking the Garage object and navigating to the GHAPropertyChangedScript menu item as shown here. The list of properties associated with the Garage object is shown. Select the Occupied property and click Ok. A new node under the Garage object will be created that contains the beginnings of a script. Highlight the OnChangeOccupied script node under Garage and a code editor will appear. In our example, the Garage contains an overhead light called OverheadLight. We want our script to turn OverheadLight to “on” when the Garage is occupied, and “off” when it is not.

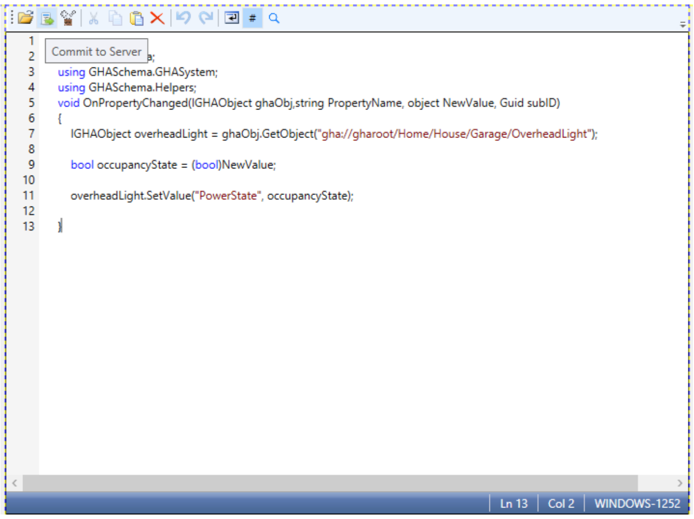
When we created the script, GHA setup a property subscription to the Garage Occupied property using the SubscribeToProperty method. It did all this behind the scenes so the user would just need to focus on the logic used to implement our desired functionality. A simple script implementing that functionality is shown here:

```csharp
using GHASchema;
using GHASchema.GHASystem;
using GHASchema.Helpers;
void OnPropertyChanged(IGHAObject ghaObj,string PropertyName, object NewValue, Guid subID)
{
    IGHAObject overheadLight = ghaObj.GetObject("gha://gharoot/Home/House/Garage/OverheadLight");

    bool occupancyState = (bool)NewValue;

    overheadLight.SetValue("PowerState", occupancyState);

}
```

Note the name of the method “OnPropertyChanged”. Any subscription to a property would need to point to a method that implements this signature. The first parameter is the IGHAObject containing the property that has changed. In this case, it is the Garage object. The second contains the name of the property – the *Occupied* property of the Garage object. The third contains the new value of the property – the *Occupied* property is a Boolean value, but the method signature declares it as a .Net Object type. We will need to cast that to a bool when we want to use it later in the script. The fourth is the internal subscription identifier used for this subscription (it is returned by the SubscribeToProperty method, but that is not important for this discussion).

The next line of code uses the GetObject method to get a handle on the OverheadLight object we wish to control. This form of the GetObject method takes the full path to the object. Note that GetObject can return a child of the current object using a relative path (e.g. just the name), or *any* object in the entire hierarchy using the full path as shown in the example. Once this line of code executes, the variable overheadLight (declared as an IGHAObject) has a reference to the OverheadLight object. That reference has full access to all properties and methods exposed by the OverheadLight object.

Line eight, casts the value of the *Occupied* property as passed to the OnPropertyChanged method into its native Boolean representation. The last line uses the SetValue method on the OverheadLight object to set the *PowerState* property to the same value as the *Occupied* property of the Garage object.



```

a;
using GHASchema.GHASystem;
using GHASchema.Helpers;
void OnPropertyChanged(IGHAObject ghaObj,string PropertyName, object NewValue, Guid subID)
{
   IGHAObject overheadLight = ghaObj.GetObject("gha://gharoot/Home/House/Garage/OverheadLight");

   bool occupancyState = (bool)NewValue;

   overheadLight.SetValue("PowerState", occupancyState);

}
```

We are not done quite yet. Note the second button in the code editor window. It is the "Commit to Server" button. <u>You must click on that to save your work.</u> Now hover over the third button. That is the "Build" button. While it is not necessary to "Build" your scripts, it is a good idea. If you have a syntax error, the editor will show you after you click on the "Build" button.

That is it, you built your first script! You also learned how to use a few of the key methods and concepts of the IGHAObject.

Now let us move on to a slightly more complicated scenario. In this scenario we want a water sensor object to notify us if it detects water via the GHA Text to Speech function on the default audio device for the GHA Server, and on Amazon echo devices (I bet you didn't know you could do this, well, GHA can!). We want GHA to remind us every five minutes via text to speech until it has detected that the water is no longer detected.

We have created a water sensor under the Home hierarchy called SumpPumpSensor. The SumpPumpSensor object has a property called WaterDetected. As in the previous example, we will create a GHAPropertyChangedScript by right clicking on SumpPumpSensor and selecting the appropriate item under "New". In this case, the script is called OnChangeWaterDetected. The script implementing the desired functionality is shown here:

```
using GHASchema;
using GHASchema.GHASystem;
using GHASchema.Helpers;
using GHASchema.Classes.Capabilities.Speech;
void OnPropertyChanged(IGHAObject ghaObj,string PropertyName, object NewValue, Guid sub
ID)
{
     bool waterDetected = (bool)NewValue;

    // Get a handle on a speech device


12.      ITTSText alexaRoot= ghaObj.GetObject("gha://gharoot/Devices/GHAAlexaRoot") as ITTSText;
13.
14.
15.   if(waterDetected)
16.     {
17.         // Water is detected
18.
19.
20.
21.         GHASys.Speak("Attention!  Water is detected by the sump pump" , alexaRoot);
22.         GHASys.Speak("Attention!  Water is detected by the sump pump");
23.
24.          // Initiate nag
25.
26.                 Action nag1 = new Action( () =>
27.                     {
28.
29.                          GHASys.Speak("Attention!  Water is detected by the sump pump");

30.                          GHASys.Speak("Attention!  Water is detected by the sump pump" , alexaRoot);
31.                     });
32.
33.                   var timer = GHASys.AddTimer("Timer_" + ghaObj.ObjectID.ToString(), 5.0*60.0*1000.0, nag1, true);
34.                   timer.Enabled = true;
35.     }
36.     else
37.     {
38.         // Water is no longer detected
39.
40.          GHASys.RemoveTimer("Timer_" + ghaObj.ObjectID.ToString());
41.
42.
43.          GHASys.Speak("Attention!   Water is no longer detected by the sump pump");
44.
45.
46.
47.
48.
49.     }
50. }
```

As in the previous example, the OnPropertyChanged method defines the entry point for the script. GHA will call it when the WaterDetected property is changed. Line 7 casts the NewValue parameter as a Boolean variable called waterDetected. Line 12 uses the GetObject method to get a reference to a device called GHAAlexaRoot. In this example, however, it converts that object to a variable defined as a GHA **Capability** called *ITTSText*. We will talk about why we do this in a moment.

Line 15 sets up a conditional section of code where we process both the "true" and "false" states of waterDetected (the variable that is currently holding the value of the WaterDetected property). Lines 21 and 22 utilize the Speak function from GHAHelpers. As we will learn in more detail later, GHAHelpers expose system-wide functions in a simplified manner. Any script, Module or device driver can access the functions in GHAHelpers. In this example, we make two calls to the Speak function. The first tells GHA to speed the specified line of text to a specific device: alexaRoot. The type accepted by this form of the Speak method is ITTSText. This is why we explicitly converted the result of the GetObject call in line 12 to ITTSText. Line 22 uses the default audio device of the GHA Server to speak the specified text. The next few lines implement the five-minute reminder requirement.

The .NET framework defines an **Action** data type which essentially encapsulates a segment of code that can be referred to by a single variable. In our example, we define an **Action** variable appropriately called *nag1*. The nag1 variable is assigned the code segment shown in Lines 29 and 30 – GHASys.Speak calls to notify users in the home that a leak has been detected. Note that the Speak methods are identical to the statements in line 21 and 22 previously described.

Line 33 introduces another GHAHelper function called AddTimer. We will explore these helper function in more detail later, but in short, the AddTimer method creates a timer of the specified name (we'll describe how we named it later) and duration (in milliseconds). The third parameter of AddTimer specifies the Action that is to be taken when the timer expires. The fourth is the auto-reset flag. When it is true, the timer automatically restarts after the timeout period has expired. This causes the Action to be executed every time the number of milliseconds in "duration" have expired. A reference to the newly created timer is assigned to the variable called timer.

We just described what happens when the *WaterDetected* property is set to true. An attention message is spoken on the default audio device of the GHA server, and the Amazon Echo devices are getting notifications alerting people in the home. These alerts and notifications will continue – every five minutes – until the *WaterDetected* property is set to false (presumably when the water issue is resolved). When it is set to false, we need to take a few actions to stop the "nagging" done by the script. That is handled in lines 36 through 43.

The timer that we created in Line 33 is deleted by another GHAHelper function called RemoveTimer. The RemoveTimer method takes the name of the previously created timer as its single parameter. Note the name we are using. We take the static text string "Timer_" and append the ObjectID of the IGHAObject that contains the *WaterDetected* property. In this case it is the SumpPumpSensor object. Using this technique, we can guarantee that the name of this timer is unique. All active timers are visible in the Modules portion of the hierarchy under a node called GHATimers.

GHA Modules

Modules provide the most flexible way to add functionality to GHA. At its core, the GHA Module functionality is a code generator. The objective of a GHA Module is to provide IGHAObject classes that may be used within the GHA hierarchy. The items you see when you right click under the Home node in the hierarchy are actually built-in IGHAObject classes that can be used to create nodes. The GHA Modules functionality allows us to add custom IGHAObject classes to our GHA Server thus extending its capabilities.

The end-user defines a module, creates classes under that module, and adds properties and methods under each class. All of this can be done through the GHA Configurator user interface. In fact, using the Modules functionality, it is possible to create a class containing only properties that can be added to the GHA hierarchy without writing a single line of code! Let us create a simple module as an example.

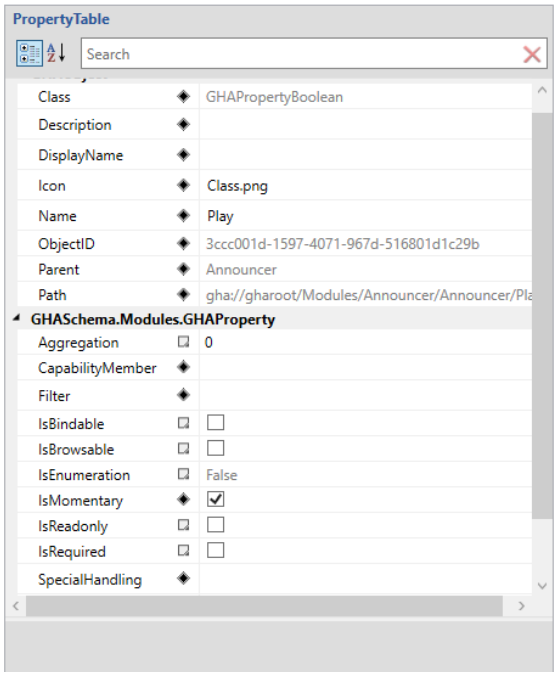
Our module will implement a class called Announcer which can be used to provide simple text to speech functionality. The first step is to create a module under the Modules node in the hierarchy. Right click on Modules, select New>GHASchema.Modules.GHAModule>GHAModule. A node will be created under

Modules called GHAModule. Right click that, select Rename and provide the new name of Announcer. Now that we have the module created, we need to create a class that can be subsequently used by GHA.

Right click Announcer and select New>GHASchema.Modules.GHAClass>GHAClass. A new class is created called GHAClass. Rename it to Announcer.

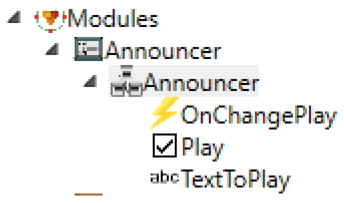
The first thing we need to do is tell GHA where in the hierarchy this class is allowed to be added as an object. We do this by using the ContainedBy function within modules. Right click on the Announcer class and select ContainedBy>Home>Home. That will instruct the GHA Executive to allow objects of the type Announcer to be created under the Home portion of the hierarchy.

Now we need to add some properties to our new class. For our example we will create two properties. The first will be a text property to enter the text to speak. The second is Boolean property which will be used to trigger the text to speak action. Right click the Announcer class again and select New>GHASchema.Modules.Property>GHAPropertyString. Rename the newly created node TextToPlay. Right click on the Announcer class again and select New>GHASchema.Modules.Property>GHAPropertyBoolean. Rename the newly created node Play. Highlight the node named Play. You will see a series of properties that will define the behavior of this Boolean property. We want this Play property to be *momentary*. That is, we set it and it resets automatically. We want it to act like a push button. To achieve this, we need to set the IsMomentary property to true as shown in Figure (xx).


PropertyTable

Class	GHAPropertyBoolean
Description	
DisplayName	
Icon	Class.png
Name	Play
ObjectID	3ccc001d-1597-4071-967d-516801d1c29b
Parent	Announcer
Path	gha://gharoot/Modules/Announcer/Announcer/Pl
GHASchema.Modules.GHAProperty	
Aggregation	0
CapabilityMember	
Filter	
IsBindable	☐
IsBrowsable	☐
IsEnumeration	False
IsMomentary	☑
IsReadonly	☐
IsRequired	☐
SpecialHandling	

Now we need to add some code to react to when the Play property changes. We do this through a GHAPropertyChange method. Create one by right clicking the Announcer class again, selecting New>GHASchema.Modules.GHAMethod>GHAPropertyChangeMethod. The names of currently provided properties are displayed. Select the "Play" property. A new node under the Announcer class will be created called OnChangePlay. The newly created Announcer module with its Announcer class will look something like Figure (xx).



The OnChangePlay method needs to have code added to take the actions required to actually perform the desired text to speech functionality. Highlight the OnChangePlay node and a code editor will appear. Before we add the code, we need to understand the context provided by GHA to the OnChangePlay object. You can assume that four variables are already present even thought you cannot see them in the editor. These include:

Variable name	Data type	Description
ghaObj	Object	The object containing the property that has changed
PropertyName	string	The name of the property that has changed
NewValue	Object	The new value of the property
subID	Guid	Subscription identifier created by the SubscribeToProperty method.

We add the following code to the OnChangePlay method using the code window.

```
1.  bool playState = (bool)NewValue;
2.  if(playState)
3.  {
4.      string message = this.TextToPlay;
5.      GHASys.Speak(message);
6.  }
```

We cast the NewValue variable to a Boolean variable called playState first. Line 2 tests if the playState is set to true. Remember that we have defined this property as momentary, so this method will be executed twice. The first time it will have a value of true, the second false. We are only interested if the property is set to true. If playState is true, we setup the call to the GHAHelper speak. We grab the string that is in the TextToPlay property we previously created and assign it to a string variable called message. We then invoke the GHAHelper function Speak to convert the message to speech and play it over the default audio device of the GHA Server.

Before we leave this node, we must commit the code we entered to the server. At the top of the code editor you will find a series of buttons. The second button will commit the code entered to the server. The final step in making the classes in our new module available to GHA is to Generate the module.

Generating the Module involves right clicking the Announcer Module node (not the class) and selecting Generate. The Generate function builds the module and instructs the GHA Executive to load it making it available for use within the system. Any changes to the module will not be made available to the GHA system until the Generate function is executed.

GHA Modules – A deeper dive

Let us take a walk-thru of the code that is generated by GHA behind the scenes. This exercise will show us more details about how IGHAObjects are used. The code generated by GHA for this module is shown below:

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.42000
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
```

```csharp
// </auto-generated>
//------------------------------------------------------------------------------

namespace GHAModules.Announcer {
    using System;
    using System.Diagnostics;
    using System.ComponentModel.Composition;
    using GHASchema.GHASystem;
    using GHASchema.Helpers;


    [System.Serializable()]
    [System.ComponentModel.Composition.Export("Announcer", typeof(GHASchema.GHASystem.IGHAPlugin))]
    [GHASchema.Attributes.GHAContainedBy(typeof(GHASchema.GHASystem.Home))]
    public class Announcer : GHASchema.GHASystem.GHAObject, System.ComponentModel.INotifyPropertyChanged, GHASchema.GHASystem.IGHAPlugin {

        private string @__texttoplay;

        private bool @__play;

        public Announcer()
            {
            _Class="Announcer";
            }

        public Announcer(string name, GHASchema.IGHAObject parent, GHASchema.GHASystem.GHARootClass ghaRoot, string objID) :
                base(name, parent, ghaRoot, objID) {
                this.SubscribeToProperty(this,"Play","OnChangePlay");
        }

        [GHASchema.Attributes.GHAProperty(Description="", IsReadonly=false, IsBindable=false, IsRequired=false, IsBrowsable=false, IsMomentary=false, SpecialHandling="", Filter="", Aggregation=Aggregation.None, IsVisible=true)]
        public virtual string TextToPlay {
            get {
                return this.@__texttoplay;
            }
            set {
                if ((@__texttoplay == value)) {
                    Debug.WriteLine("Value did not change");
                }
                else {
                    this.@__texttoplay = value;
                    base.OnPropertyChanged("TextToPlay");
                }
                Debug.WriteLine("Property Setter invoked");
            }
        }

        [GHASchema.Attributes.GHAProperty(Description="", IsReadonly=false, IsBindable=false, IsRequired=false, IsBrowsable=false, IsMomentary=true, SpecialHandling="", Filter="", Aggregation=Aggregation.None, IsVisible=true)]
        public virtual bool Play {
            get {
                return this.@__play;
            }
            set {
                if ((@__play == value)) {
```

```
62.           Debug.WriteLine("Value did not change");
63.         }
64.         else {
65.           this.@__play = value;
66.           base.OnPropertyChanged("Play");
67.         }
68.           Debug.WriteLine("Property Setter invoked");
69.       }
70.     }
71.
72.     public virtual void OnChangePlay(object ghaObj, string PropertyName, object New
    Value, System.Guid subID) {
73.       try {
74.           bool playState = (bool)NewValue;
75.
76.           if(playState)
77.           {
78.             string message = this.TextToPlay;
79.             GHASys.Speak(message);
80.
81.             message = announcer.GetValue("TextToPlay") as string;
82.
83.             GHASys.Speak(message);
84.
85.
86.
87.           }
88.       }
89.       catch (System.Exception ex) {
90.         // Handle exceptions here
91.         var debugEvent = GHASys.BuildDebugEvent(this, ex);
92.         GHASys.FireDebugCallbacks(debugEvent);
93.       }
94.     }
95.   }
96. }
```

There are a few observations from the code that yields details on building objects for GHA. Line 22 declares the Announcer class. Note that in .NET terms it inherits from GHAObject and implements the InotifyPropertyChanged interface. To be compatible with the GHA Hierarchy, a class must inherit from GHAObject. While the Announcer class also directly implements INotifyPropertyChanged, this is not strictly necessary because GHAObject already implements that interface. Also note that the Announcer class is decorated with the [Serializable] and [ContainedBy] attributes. The [Serializable] attribute is required to support various GHA internal functions. The [ContainedBy] attribute instructs the GHA Configurator to show the class as an available child to objects of the specified type: The Home class in this case.

The constructor signature for the class in line 33 is unique to classes declared by Modules. Because GHA Modules are loaded in separate .NET Application Domains, the GHA Executive must treat them slightly differently hence the need for this unique constructor signature. When you develop Device drivers for GHA, you will use the standard constructor signature. More on that when we talk about Device drivers.

Notice within the constructor there is a call to the SubscribeToProperty method. In this example, we are subscribing to the “Play” property of the current class and telling the class to execute a method called “OnChangePlay” when the “Play” property is changed. Remember when we added the

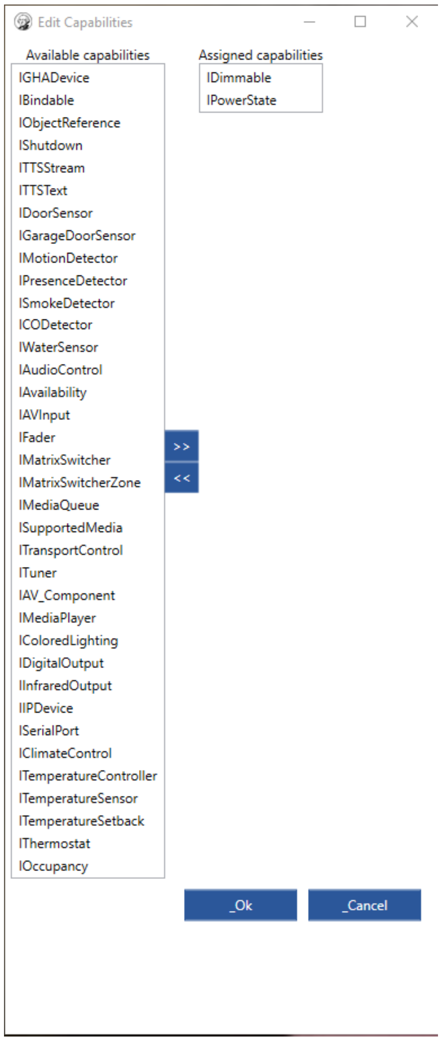
GHAPropertyChange object in our example? This line of code actually implements the property subscription.

Lines 39 through 70 declare the two properties we defined in our example: TextToPlay and Play. Notice that the code generator takes care of creating getters and setters for each property. Pay special attention to the decoration of the "Play" property on line 55. The GHAProperty attribute describes any special handling that is required for this property. Notice the variable called "IsMomentary" in the attribute parameter list. It is set to *true*. Remember in our example where we declared the "Play" property as momentary? This is where it is actually implemented. Keep this trick in mind when we get into developing device drivers.

The last piece of our walk-thru is the "OnChangePlay" method starting in Line 72. Most of the code in that method should look familiar. We created it within the editor. The GHA Modules subsystem wrapped our custom code within a method of the proper signature for handling property changes and included an exception handler to ensure any problems with our custom code would not cause the GHA System to crash.

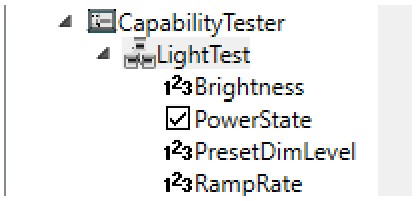
Adding Capabilities

GHA **Capabilities** can be added to GHA Module classes by right clicking the class and selecting Add/Remove capabilities. The Edit Capabilities window will appear showing available capabilities on the left and assigned capabilities on the right. Let us create a class that will be used to implement a dimmable light. We will not repeat the steps needed to create a new module and class here. Our new class is called LightTest. Note that in Figure (XX) we have already selected the IDimmable and IPowerState **Capabilities**. After clicking the Ok button, GHA will populate the LightTest Class with the properties and methods associated with the assigned capabilities. See Figure (xx).


Edit Capabilities

Available capabilities
IGHADevice
IBindable
IObjectReference
IShutdown
ITTSStream
ITTSText
IDoorSensor
IGarageDoorSensor
IMotionDetector
IPresenceDetector
ISmokeDetector
ICODetector
IWaterSensor
IAudioControl
IAvailability
IAVInput
IFader
IMatrixSwitcher
IMatrixSwitcherZone
IMediaQueue
ISupportedMedia
ITransportControl
ITuner
IAV_Component
IMediaPlayer
IColoredLighting
IDigitalOutput
IInfraredOutput
IIPDevice
ISerialPort
IClimateControl
ITemperatureController
ITemperatureSensor
ITemperatureSetback
IThermostat
IOccupancy

Assigned capabilities
IDimmable
IPowerState


CapabilityTester
LightTest
Brightness
PowerState
PresetDimLevel
RampRate

Inheriting from other objects

Inheritance is a powerful feature in Object Oriented programming. It allows a new class to take on (or inherit) the capabilities from an inherited class without copying and pasting code (or even having visibility to the code of the inherited class). The GHA Modules subsystem allows a class to inherit from the wide range of GHA built-in classes. The default inherited class for custom classes in the GHA Modules subsystem is the GHAObject. To inherit from a different class, right click on the GHA Module Class and select InheritFrom. A menu of class categories will appear where you can navigate to the class you wish to inherit.

Device Drivers

The device subsystem in GHA is designed to be extensible. New devices can be created by developing device drivers that "plug in" to the GHA system. There are a few prerequisites that are needed before you can develop a device river for GHA.

Prerequisites

- Knowledge of programming in Microsoft's .NET 6.
- Visual Studio 2022 or greater (community edition is okay)

Setting up your development environment

The best way to develop for GHA is to use Visual Studio 2022. Device drivers in GHA are implemented as .NET 6 class libraries. Your first step is to setup and configure Visual Studio to build drivers for GHA.

It is a best practice to develop drivers in separate class libraries. However, there is nothing preventing you from having more than one driver in a single library. It is assumed that the reader has a working knowledge of Visual Studio 2022, so we will not repeat the detailed steps to create a class library project. While you are creating the class library here are a few configuration items to keep in mind.

- The class library must use .NET 6 as its target framework.
- Include a reference to GHASchema.dll in your project
- Add the NuGet package Log4Net if you intend to use the GHA Executive Logging function in your driver (Very useful to debug your driver)
- Use a non-production instance of your GHA system while you develop your driver
- Install the Visual Studio Remote Debugger on your target development system

IGHADevice Interface

All GHA devices must implement the IGHADevice Interface. Classes that implement this interface (when instantiated) appear under the Devices root of GHA. It is important to note that ONLY classes destined to appear under the Devices root should implement this interface. You may note that many of the pre-built devices provided with GHA have objects that appear under device driver objects. These objects should not implement IGHADevice but should implement IGHAObject (or preferably inherit from GHAObject). A summary of that interface is shown here.

Properties

	Name	Description
	Version	Returns the version number for the device driver

Methods

	Name	Description
	Initialize	Called by the GHA executive during device initialization. Perform device-specific initialization in this method
	Install	Called by the GHA executive when the device driver is installed
	Shutdown	Called by the GHA executive during shutdown operations. Perform device-specific shutdown steps in this method
	Uninstall	Called by the GHA executive when the user deletes a device driver from the Devices node

GHA Executive interaction with device drivers

On startup, the GHA Executive searches the Devices folder under the installation path for class libraries that implement the IGHADevice Interface. It then loads the class library and makes it available for subsequent loading in GHA. If the device has already been installed, the *Initialize* method is invoked. It is expected that any startup-specific activities for the device are performed in this method. If the device has not been installed, the executive ignores the device specified in the library. However, it does make any classes that implement the IGHADevice interface available when the user right-clicks and selects "New" under the Devices node in the GHA Configurator.

When a user decides to create an instance of a device, they simply right-click the Devices node in the GHA Configurator and click "New". A list of all available device classes is presented. When the user selects one of these device classes, the GHA Executive creates an instance of the class and invokes its *Install* and *Initialize* methods.   The *Install* method performs any device-specific one-time steps that are required by the device driver. For example, a device driver might need to create objects to manage multiple instances of a specific device type such as audio devices installed on a computer. In that case, the Install method might search the local computer for audio devices and create an object for each device.

During GHA shutdown, the GHA Executive calls the *Shutdown* method for each currently installed device driver. Any device-specific shutdown steps should be performed in this method.

Once the device is installed and initialized, the GHA Executive allows the device driver to manage itself. That is a key concept to understand. The device driver has full access to all IGHAObjects in the system. It can also inadvertently monopolize processing resources. Just a few things to consider when developing drivers.

- Run the device management on a new thread created during the device initialization process. That will keep blocking operations localized to the device driver.
- Be careful of using blocking calls in the *Install* or *Initialize* methods.
- Ensure you call OnPropertyChanged(string name) method in your property setters when the value of the property has changed to ensure that property subscriptions are fired. Alternatively, using an MSIL weaver like Fody.PropertyChanged can be used to simplify your code.
- Ensure you have exception handling built into your device driver. While the GHA Executive tries to protect itself from misbehaving device drivers, it is possible that exceptions might "bubble up" from a faulting device driver crashing the entire GHA system.

When the user decides to delete a device, the GHA Executive invokes the driver's *Uninstall* method. That method should, at a minimum, gracefully shutdown the device driver. Additionally, the *Uninstall* method should remove any references in the hierarchy to GHAObject types defined within the driver. For example, a media-handling device might create a device-specific type that is used in the GHA Media subsystem. References to those types must be removed when the device driver is deleted or there may be impact on the GHA system.

A simple example

In this example, we will build a "stub" device driver to handle lighting control. The example itself does not implement control on a real device. Rather, it shows a simple device model that implements a few key features of GHA: Implementing a base device driver, implementing device capabilities, making objects Bindable.

Device Capabilities in GHA are actually .NET Interfaces. In the Modules section, we simply did point-and-click definition of capabilities to be implemented by classes in the module. Building device drivers is a little more complex in that we must provide code to implement .NET interfaces. Thankfully, Visual Studio provides the ability to pre-populate properties and methods defined by .NET interfaces with templates. That allows us to focus on the logic of the device driver versus the structure to make it compatible with GHA.

In our lighting control example, we want to implement a driver that implements the *IPowerState* and *IDimmable* capabilities. We do this by declaring that the class must implement the IPowerState and IDimmable interfaces. Further, this device would be of much use if we did not allow it to bind to a device model under the Home hierarchy, so we must also implement the IBindable interface. Keep in mind that all classes used directly in the GHA hierarchy must also inherit from IGHAObject. The source code for our example is shown here.

```csharp
using GHASchema;
using GHASchema.Attributes;
using GHASchema.Classes.Capabilities;
using GHASchema.Classes.Capabilities.Lighting;
using GHASchema.Classes.Capabilities.Power;
using GHASchema.GHASystem;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GHADevices.LightStub {
  [Serializable]
  public class LightStub: GHAObject, IGHADevice, IPowerState, IDimmable, IBindable {
    public LightStub(string name, IGHAObject parent, string ObjectID = ""): base(name,
parent, ObjectID) {

    }
    private string _version = "1.0";
    public string Version => _version;

    private bool _powerState = false;
    [GHAProperty(description: "Power state, On if true, Off if false")]
    public bool PowerState {
      get => _powerState;
      set => _powerState = value;
    }

    private double _brightness = 0.0;
    [GHAProperty(description: "Level of lighting between 0.0 and 1.0")]
    public double Brightness {
      get => _brightness;
      set => _brightness = value;
    }

    private double _presetDim = 1.0;
    [GHAProperty(description: "Level to set lighting when the PowerState is true betwee
n 0.0 and 1.0")]
    public double PresetDimLevel {
      get => _presetDim;
      set => _presetDim = 1.0;
    }

    private double _rampRate = 0.1;
    [GHAProperty(description: "Rate in seconds at which to get desired lighting level")
]
    public double RampRate {
      get => _rampRate;
      set => _rampRate = value;
    }

    IGHAObject _boundObject = null;
    public IGHAObject BoundObject {
      get => _boundObject;
      set => _boundObject = value;
    }

    public object Initalize() {
      return true;
    }

    public object Install() {
      return true;
    }

    public object Shutdown() {
      return true;
    }

    public object Uninstall() {
      return true;
    }
  }
}
```

You can see in the class constructor that we declare this class as inheriting from GHAObject (the recommended implementation of IGHAObject). It also defines the IGHADevice, IPowerState, IDimmable, and IBindable interfaces. You can also see that we have declared the class as Serializable. This is required. This allows the GHA Executive to save the state of the device driver between restarts.

As we further examine the example, we can see the properties and methods associated with the various interfaces we have declared for this class. Since this is a simple example, the IGHADevice methods simply return a true. Technically, we can return anything as these values are ignored by the GHA Executive. What is more interesting is how we structure the property definitions.

It is a best practice in GHA to define your own backing store variables for properties. While not strictly required, it aids in debugging. Also please note the GHAProperty decoration on each of the defined properties. This decoration tells GHA if there is any special handling required for this property. At a minimum, you should define a description in the GHAProperty decoration. That description will show up in the GHA Configurator making it simpler for users to understand the function of the property.

Device Templates

GHA offers a number of device templates that can be inherited in your custom device drivers. Much of the functionality required to build a device driver is already included in the device templates. Where device-specific processing is required, the device template implements either a virtual or abstract method to allow the developer to build their own device. These classes are browsable using the built-in Visual Studio Object browser. Online documentation for device templates and GHA **Capabilities** can be found here.

Matrix Switcher

The Matrix switcher template is used by any device that can associate an input to an output. The Nuvo Concerto driver uses the Matrix switcher template. The Matrix switcher template can also be used to define other types of devices that can switch between various inputs. For example, a Television.

GHAMatrixSwitcher

GHAMatrixSwitcherInput

GHAMatrixSwitcherZone

Media Player

GHAMediaPlayerBase

Serial Port

GHASerialPortBase

Tuner

GHATunerBase

Television Tuner

GHATelevisionTunerBase

UPnP Device

GHAUPnPDeviceBase

UPnP Media Renderer

GHAUPnPMediaRenderBase