Interfacing with Status Enterprise’s Server using the provided .NET SDK

1. Including necessary references

Before even thinking about interacting with the Status Enterprise server, it is necessary to include the references necessary to grant access to the classes needed to work with Status Enterprise. This guide will be written for C# in Visual Studio 2015, but any .NET language should work.

On the right-hand side of the screen, after opening a new project, you should see the Solution Explorer pane; in it, a “References” drop-down. Right click on this, and select Add Reference. Click “Browse” at the bottom of the window, and navigate to the location where B-Scada/Status Enterprise is installed (probably C:\Program Files\B-Scada\StatusEnterprise). There are several different .dll’s you can reference, but some that you should select are Status.Common.Diagnostics.dll, Status.Common.Interfaces.dll, Opc.Ua.Core.dll, and Status.DataModel.ObjectModel.dll. After selecting these, the corresponding assemblies need to be checked of (added as references) in the Reference Manager window; go to Extensions on the left side of the window and scroll down until you see the corresponding entries (they should have identical or very similar names, without the file extension), and check them off.

1. Including the necessary “include” or “using” statements.

So now the necessary files are linked to the project, but they still aren’t referenced *from* within the program files. In C#, with the other “using” statements, you’re going to want to include statements for Status.DataModel.ObjectModel, Opc.Ua, Status.DataModel.ObjectModel.Common, and Status.Common.Diagnostics.

Note: if there are other classes you need, and which you know should be included in the SDK, they can usually be found (if the necessary .dll files have been referenced, as detailed in step 0) using the Object Browser. If you can’t figure out how to access the Object Browser, right click on one of the entries under References in the Solution Explorer pane, and select ‘View in Object Browser’. Here there is a search function to help you find which paths need to be included in your code with a “using” statement in order to access the required functions.

1. Making the Connection

So now that you have the classes you need accessible, it’s time to make the connection to Status Enterprise’s server. The end goal is to have a StatusObjectModel object, and there are 2 ways of doing so:

* 1. First is the ‘headless’ version, which doesn’t require a human operator to enter login information to a GUI on runtime. This method uses a StatusServerClient object instance. The Tooltip for the class instantiation should help with determining what each argument should be; most of the information should correlate with the Status Enterprise server, such as the username and password set up when the server was installed, or the network location of the server. Most of the fields (including the optional ones) have analogs to fields (some being listed under “Advanced Settings”) in a regular login window (such as when you log in to one of Status Enterprise’s apps, like the Data Model Designer), which can give you an idea of what values to put in each one. After the object is created, the instance function StatusServerClient.Connect() should be called.
  2. The second method is somewhat less work to implement, but is only suitable for creating a client application for which there is a human operator available to input login data. This method uses a ServerLogon object, along with the instance method ServerLogon.Logon(), which returns a StatusServerClient object

The last step is to actually get the object model, with the instance function StatusServerClient.GetObjectModel(), with an output parameter specifying the StatusObjectModel instance, and returning an SResult object (e.g. **SResult sresult = \_statusServerClient.GetObjectModel(out ObjectModel);**)

The object returned from StatusServerClient.Connect() and StatusServerClient.GetObjectModel() is an instance of a class B-Scada defined, SResult, to contain status information about the success or failure of the function returning it. Generally, if a function is unsuccessful, SResult.IsNotGood() will return true for the SResult object returned by the function.

\*\*For functions where an SResult object is returned, it is best practice to check to see if SResult.IsNotGood() afterward, as this can indicate any of a number of issues, and the function may return successfully, without crashing the program, but without executing as intended, which can cause hard-to-trace errors or crashes further down the line. The SResult.Message field provides helpful information regarding what went wrong\*\*

1. Interacting with the server: finding fields to write to/read from

Now that a connection to the server has been established, it makes sense to actually interact with the data held on the server. The first step in doing so is to get handles for the specific data fields we want to interact with. There are several methods for doing so, however I will primarily discuss one of them. The first, and arguably simplest, is by using the GetNodeIdFromBrowsePath function, called from the StatusObjectModel instance. The first argument is a string representing the browse path of the server value in question; this can most easily be obtained by viewing the data model designer, selecting the object in question, and looking in the lower right-hand corner of the window (see below); the field titled Browse Path is what you need, with only one addition: the property name itself is not listed in the browse path, so if I wanted to select the “Command” field of Printer001, I would need to add “/Command” to the end of the browse path below

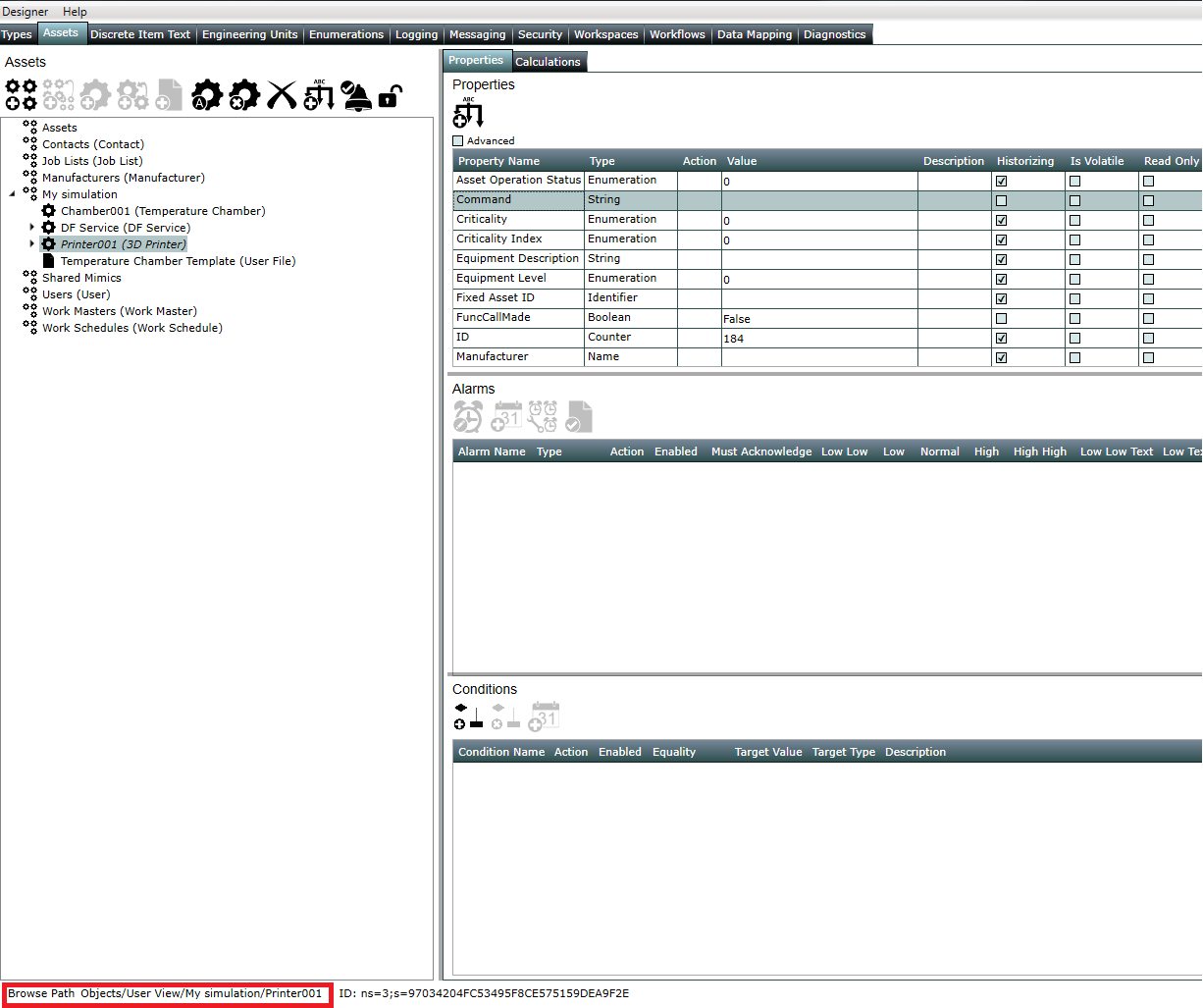


Figure 1: Status Enterprise Data Model Designer with Browse Path boxed in red

The second argument is an output variable, a NodeId object, which serves as a handle used by many of the provided functions in interfacing with the server.

If you want to perform some action on multiple fields in an asset, or on multiple assets, this can be done by casting the ObjectBase instance to a View type, and then parsing view.Components, which holds ObjectBase objects

* 1. Getting data from the server

So now that we have the NodeId, we need to actually get an object from that, and read the value of that object. This is done with the ObjectFromNodeId() function, called from the StatusObjectModel instance. The first argument is the NodeId, and the second is an output argument of an ObjectBase instance. The ObjectBase class basically serves as an interface between the data retrieved from the server and .NET objects, since the ObjectFromNodeId function doesn’t intuitively know what type of data is coming back from the server. It is up to the programmer to cast the value of the ObjectBase object to the appropriate type (a simple cast will work, e.g. ObjectBase x 🡪 double x2=(double)x.Value)

* 1. Writing data to server

Writing data to the server doesn’t require any other intermediate types; the WriteValue function called from a StatusObjectModel instance can be used to write directly to a server field referenced via NodeId. The NodeId object is passed as the first argument; the second argument represents which attribute of the entry on the server will be written to (13 is for value, which is likely what you will use most often, but every attribute in the data entry can be written to), and the third argument is a status code. The documentation seems to be unclear on this parameter, but I have had no reason to pass anything other than 0 here. The fourth argument is the object data to be written to the server field (any object can be passed here, a specific type is not necessary), and the last field is a time stamp, usually DateTime.Now

The other primary method for interfacing with the server is using subscriptions; I, however, have not used subscriptions in my work, and so the only information available is that in the documentation (see *Status Enterprise User Guide (Part 10) - Object Model Reference, page 24*)