

Simio API Note: Emulating World-Time

Creation: June 2018 (Dhouck)

Revisions: Jan 2018 (Dhouck) – Added to Github

Revisions: Mar 2023 (Dhouck) – Updates

Contents

Simio API Note: Emulating World-Time	1
Overview	2
Some Background Information on the Simio Engine	3
Extending Entities with the API	4
World-Time Emulation	4
External Device Communications in World-Time	5
How it Works: User Step Code	8
EmulationSync Step	9
Device Step	10

Overview

This API Note describes how Simio can be configured to accomplish these two tasks:

1. Emulating World-Time; that is, running the Simulation so that each second of simulation time (Sim-Time) corresponds to 1 second of World-Time (or, if you prefer, Real-Time or Wall-Clock-Time).
2. Communicating with an external device (external to Simio), which is also in World-Time.

Note: (March 2023) The Simio product now includes features that permit world-time emulation. This is dependent upon which license you are using. See the Simio documentation for more information. This example is still relevant as it shows alternative techniques for accomplishing world-time synchronization with external devices.

In the more general sense, this emulator allows the simulation to proceed at a time that is a factor of World Time. When this factor is 1.0, then the simulation is running exactly at World Time, and when it is 2.0, it is running at twice World Time.

This project also uses the singleton Entity Data technique (refer to Entity Data Handling document) for keeping track of Entity data.

This Note describes some complex programming topics. It assumes that the reader is familiar with C# and .NET technologies such as threading, asynchronous operations, and the Singleton pattern of programming.

Some Background Information on the Simio Engine

The Simio engine (which we'll just call the Engine) is the logic that implements the simulation and planning logic of Simio. When used as a desktop application, it usually includes a UI.

Generally speaking, a simulation run within the Engine is single threaded, meaning that any delays incurred by an Entity affect the whole run. For example, if an Entity decided to do a long-running operation (such as a large synchronous read or write to an external device), the Simio Engine (including the UI) will “hang” for the duration of that operation.

Note: We are here talking about a single “run”. When Simio is doing things such as Experiments, it scales by using a separate thread for each Experiment.

Simio provides the run-time API which allows users to create custom Steps (User Extensions). These are implemented as calls to .NET methods that the user can write and are implemented by conforming to the published Interfaces. As such, anything that can be done in .NET can be done in these steps. When a Step is executed it is done by a Token, which is a proxy for an Entity (i.e. the green triangles that are seen flowing through the Simio model). These Entities are born, they travel through the simulation, and they are destroyed.

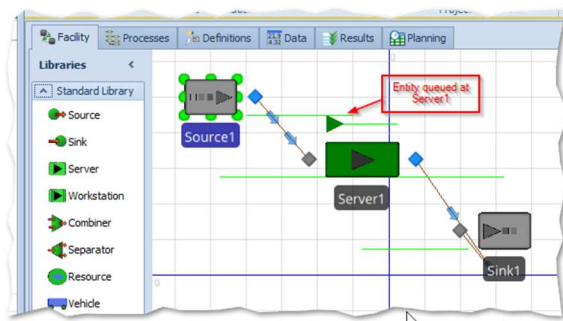


Figure 1 - The Source-Server-Sink Model

Entities are instances of Simio Intelligent Objects, and can share data using the normal Simio mechanism of Properties, State variables, etc. They can also execute logic as they travel through the system using Simio expressions.

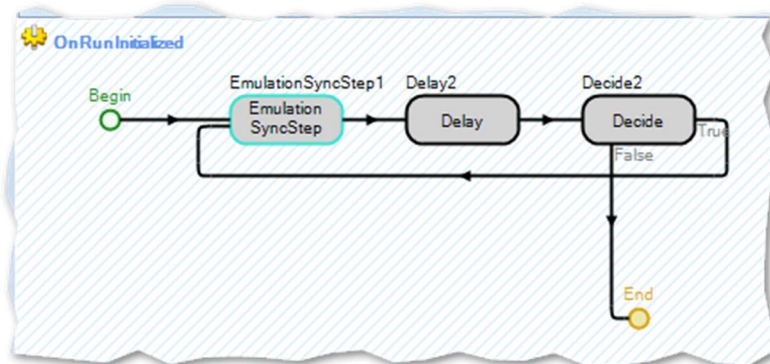
In this document we'll also show how the User Extension (User Step) can employ the C# singleton as another way that Entities can share data.

Extending Entities with the API

Although Simio does have a baked-in way of running in World-Time, it is dependent upon your license type. This example shows how it can also be accomplished with a few custom User Steps.

World-Time Emulation

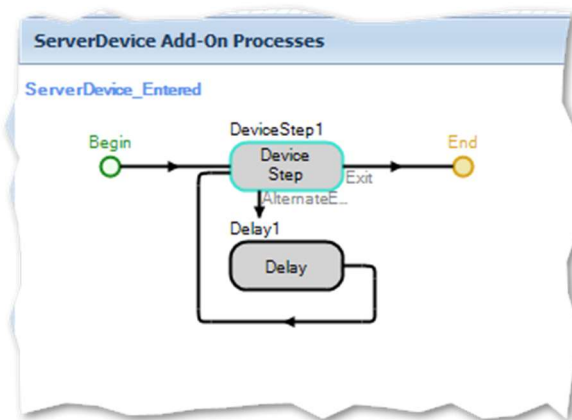
The World-Time emulation is achieved using the EmulationSyncStep, which is run at the OnRunInitialized event in conjunction with a Delay and Decide step.



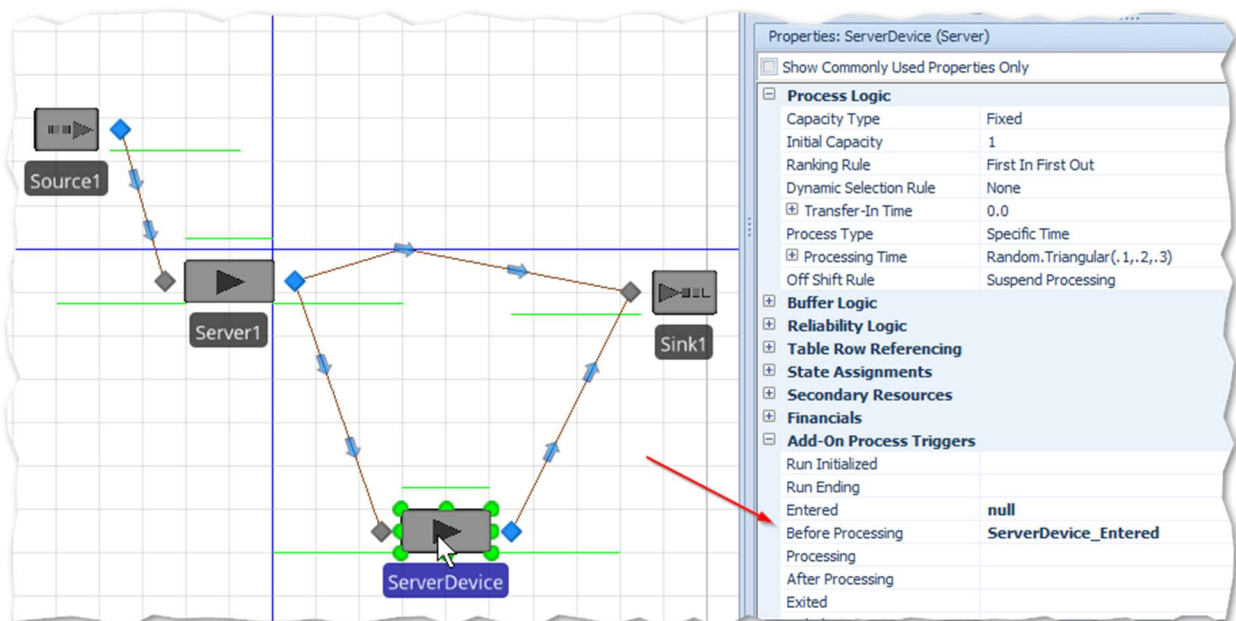
The logic for the step is quite simple: At each increment of Sim-Time (determined by the small Delay step; for example, 1 second) and if the Emulation SyncStep determines that the Sim-Time now exceeds World-Time (which is usually the case), then the simulation is paused for the difference. This loop runs for the duration of the simulation.

External Device Communications in World-Time

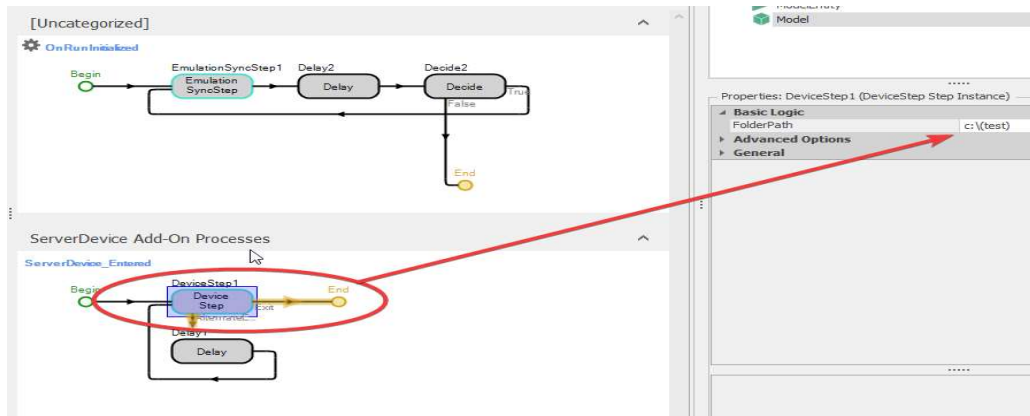
The logic for communicating with a device in World-Time involves a User “Device Step” that upon first entry requests information from the device. It then checks whether the device has responded. If there is no response, then a small Simulation Delay is called, and the Response is looked for again (a technique often referred to as “polling”). This communication protocol is called Request-Response; that is, we Request something from a device, and then expect to see a Response from it.



To illustrate these techniques, a very basic SourceServerSink (SSS) model was created and modified to have an extra “ServerDevice”.



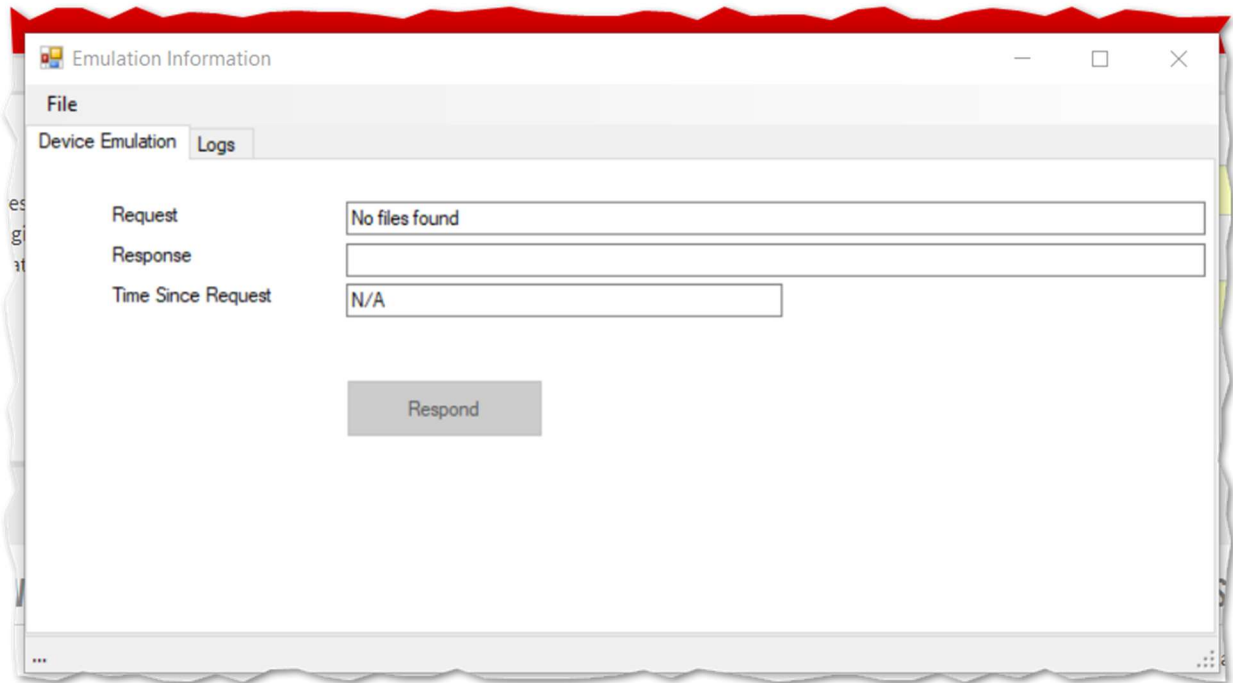
Since the example “Device” employs files for communications, a Property FolderPath exists on the Device Step, and defaults to “C:\(test)”. You can set this to any location, but it must match the setting for the DeviceEmulationProgram, which is found in DeviceEmulator.exe.config.



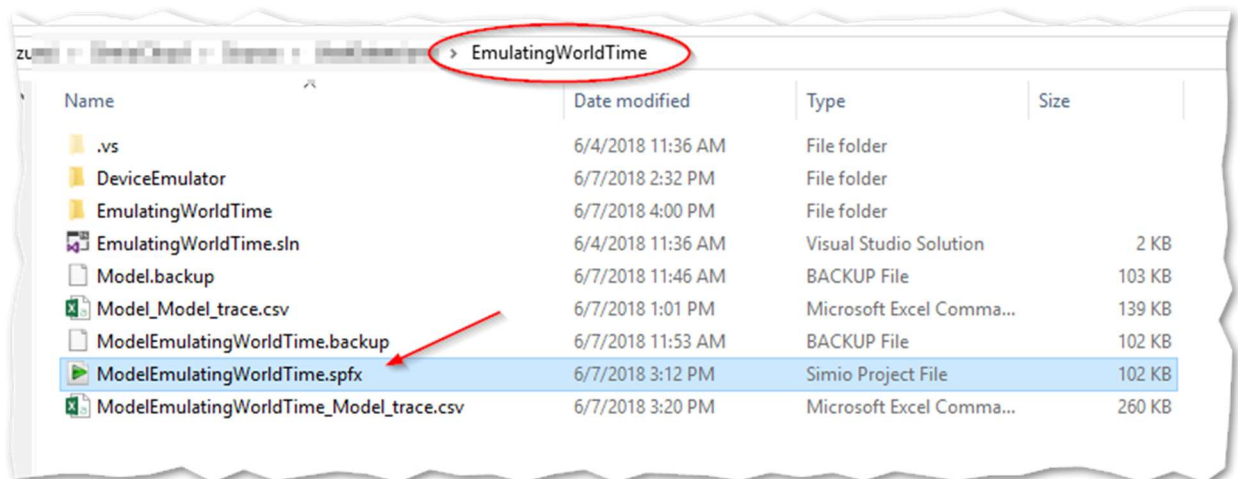
Keep in mind that at this point this example project was built for demonstration purposes. For production purposes it would need to be ruggedized (e.g. better error handling, startup, and shutdown).

For this project there is also a separate DeviceEmulator program. It is a simple WinForms application as included in the project folder.

There is also an animated Gif (SimioEmulation1.gif) which demonstrates what you should see when you run the project and an example use of the DeviceEmulator as well.



The Simio Model (ModelEmulatingWorldTime.Spfx) resides directly under the EmulatingWorldTime folder.



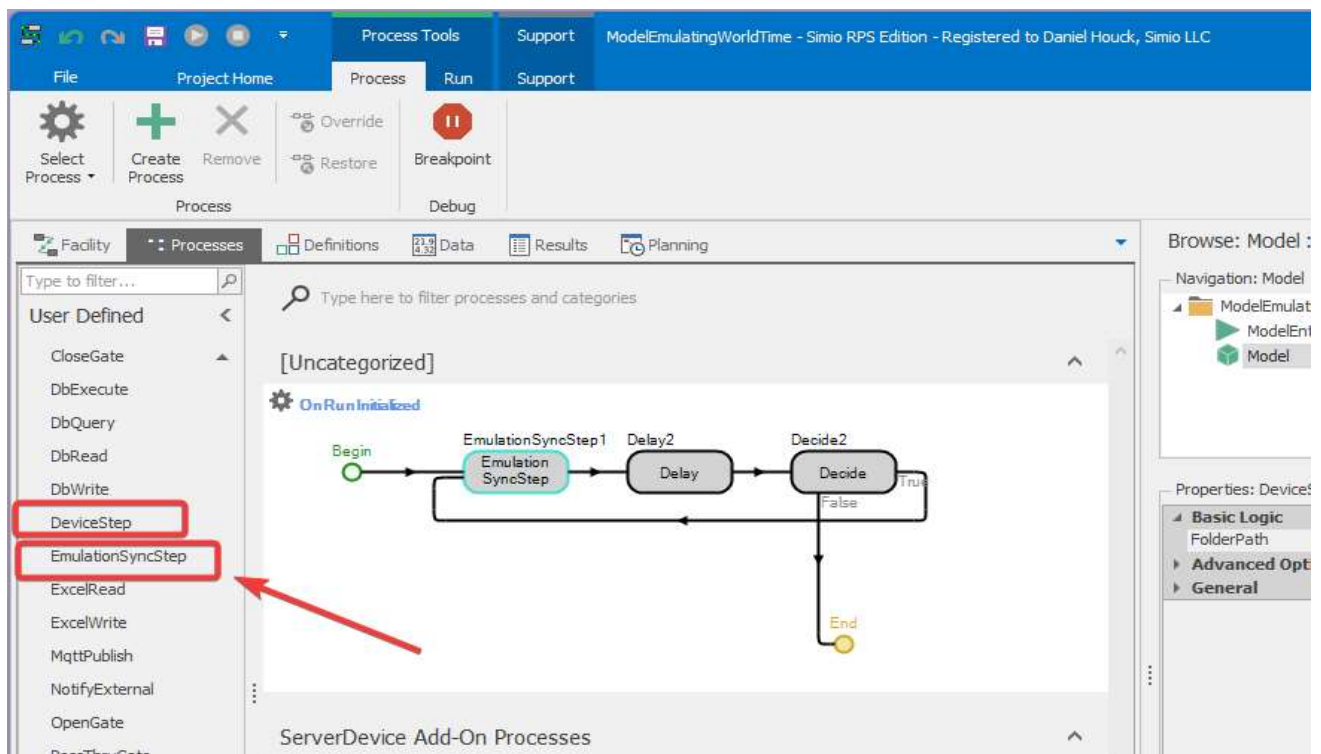
How it Works: User Step Code

This section explains how the underlying C# code within the custom (user-defined) steps works. Armed with this knowledge you can customize the steps for your particular purpose.

Again, there are two User Steps:

1. EmulationSync Step, and
2. Device Step

When you build EmulatingWorldTime project a single DLL is created. This DLL contains both of these steps, and this DLL must be located where the Simio application can find it (such as c:\users\{your-name}\documents\SimioUserExtensions). You'll know this is successful when you run Simio and go to the User Defined Steps of the Processes tab and locate the DeviceStep and EmulatorSyncStep:



EmulationSync Step

The EmulationSync Step uses a singleton to store Sim-Time and World-Time data. This could be done with Simio objects (Properties and States) if you preferred.

On start, there is a one-time store of the starting Sim-Time and World-Time.

At each run, the difference (delta) between the Sim-Time and World-Time is performed, and a "Thread.Sleep" is called for that duration to suspend the thread (which is the simulation thread).

```
94  /// <summary>
95  /// Method called when a process token executes the step.
96  /// </summary>
97  public ExitType Execute(IStepExecutionContext context)
98  {
99      EmulatingWorldTimeSingleton eContext = EmulatingWorldTimeSingleton.Instance;
100
101      // One time initialization
102      if ( eContext.Ticks == 0 )
103      {
104          eContext.StartSimHour = context.Calendar.TimeNow;
105          eContext.StartWorldTime = DateTime.UtcNow;
106      }
107
108      double simNow = context.Calendar.TimeNow; // Get Sim-Time
109
110      DateTime worldTimeNow = DateTime.UtcNow;
111      double deltaWorldSeconds = worldTimeNow.Subtract(eContext.StartWorldTime).TotalSeconds;
112
113      // Usually, simulation time will be ahead of (faster than) world time, so we'll slow it down.
114      double deltaSimSeconds = (simNow - eContext.StartSimHour) * 3600.00;
115
116      double adjustmentSeconds = deltaSimSeconds - deltaWorldSeconds;
117
118      if (adjustmentSeconds > 0)
119      {
120          // This will suspend the thread for the given number of milliseconds, which
121          // allows the simulation time to sync with world time.
122          System.Threading.Thread.Sleep((int)(1000.0 * (adjustmentSeconds)));
123      }
124      else // This would likely only happen during debugging
125      { }
126
127      eContext.BumpSimTime();
128
129      return ExitType.FirstExit;
130  }
```

Note that if you are holding up time by stopping the code in the debugger, when you finally run without breakpoints, the simulation will proceed at its full rate until it catches-up to the world time. At which point it will continue to be synced with World-Time.

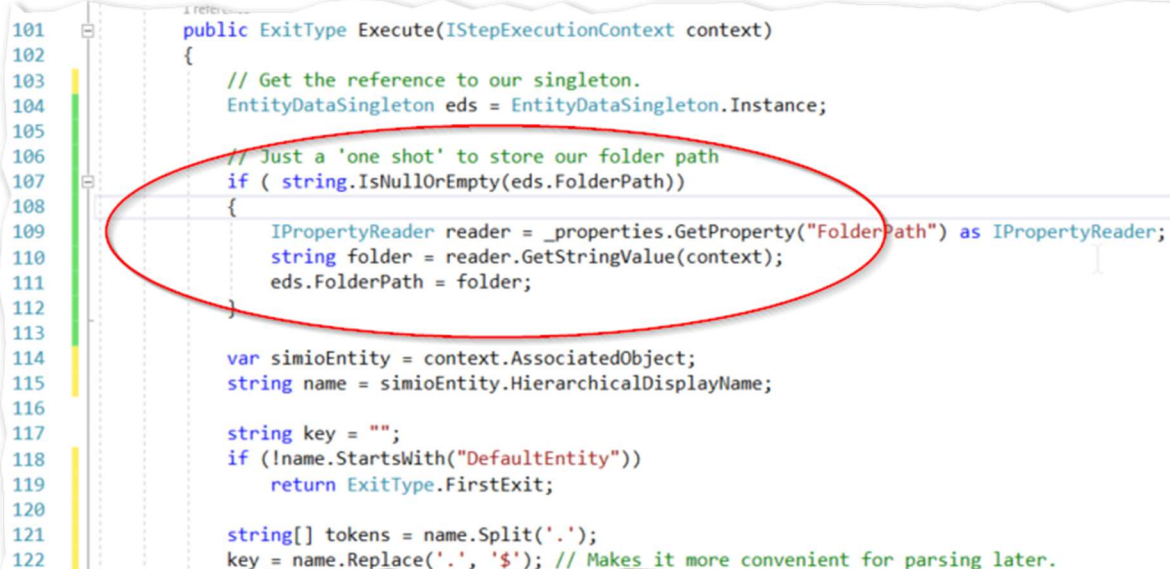
Device Step

The emulation of a device uses an external program (Device Emulator) and the file system to demonstrate a Request-Response type of device.

This step also uses a singleton structure to store data, but it was added mostly as a demonstration of how data (even large amounts of data) could be associated with an Entity. Note that the unique name of Entities makes it a rather convenient “key”.

It begins with an Entity entering the special “DeviceServer”

There is a need to differentiate whether this is our first time into the step. This is done by checking if we already have built a Request file.



```
101 public ExitType Execute(IStepExecutionContext context)
102 {
103     // Get the reference to our singleton.
104     EntityDataSingleton eds = EntityDataSingleton.Instance;
105
106     // Just a 'one shot' to store our folder path
107     if ( string.IsNullOrEmpty(eds.FolderPath))
108     {
109         IPropertyReader reader = _properties.GetProperty("FolderPath") as IPropertyReader;
110         string folder = reader.GetStringValue(context);
111         eds.FolderPath = folder;
112     }
113
114     var simioEntity = context.AssociatedObject;
115     string name = simioEntity.HierarchicalDisplayName;
116
117     string key = "";
118     if (!name.StartsWith("DefaultEntity"))
119         return ExitType.FirstExit;
120
121     string[] tokens = name.Split('.');
122     key = name.Replace('.', '$'); // Makes it more convenient for parsing later.
123 }
```

If the Request file does not exist, then we'll build one.

Regardless, there is a check for a Response file. If it exists we have our answer from the Device, so we can take FirstExit.

If it doesn't exist (the Device has not yet responded), we take door #2 (AlternateExit) which (in our Process) leads us to a short Simio-Delay step and then back again to this Device Step.



```
mulatingWorldTime EmulatingWorldTime.DeviceStep Exe
120
121 string[] tokens = name.Split('.');
122 key = name.Replace('.', '$'); // Makes it more convenient for parsing later.
123
124 // Fetch or create our entity data that will accompany the entity.
125 // This data is stored in the singleton's dictionary, and fetched by key.
126 EntityData eData = null;
127 if (!eds.EntityDataDict.TryGetValue(key, out eData))
128 {
129     eData = new EntityData(key);
130     eds.EntityDataDict.TryAdd(eData.Key, eData);
131 }
132
133 eData.TimeRequestMade = DateTime.UtcNow;
134
135 string folderPath = eds.FolderPath;
136 string requestFilePath = Path.Combine(folderPath, $"Request-{key}.txt");
137
138 // If the request file is there, then we are waiting for a response, so
139 // don't generate another request
140 if (!File.Exists(requestFilePath))
141 {
142     logit(EnumLogFlags.Information, $"Request file written to={requestFilePath}");
143
144     // Put a request file, and then we'll poll for the response.
145     File.WriteAllText(requestFilePath, "(data body: info the device might want)");
146 }
147
148 // Look for the Response file. If found, then Exit First, if not, then exit Alternate
149 string responseFilePath = Path.Combine(folderPath, $"Response-{key}.txt");
150
151 if (File.Exists(responseFilePath))
152 {
153     string contents = File.ReadAllText(responseFilePath);
154     File.Delete(responseFilePath);
155
156     logit(EnumLogFlags.Information, $"Found file={responseFilePath} Contents={contents}");
157     return ExitType.FirstExit;
158 }
159 else
160 {
161     return ExitType.AlternateExit;
162 }
```