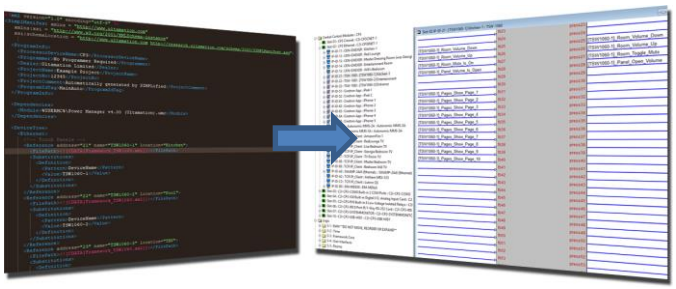
# SIMPLified 2's SIMPL Program Generator

## WHO IS THE INTENDED AUDIENCE?

SIMPLified 2's program generation feature provides a powerful tool which saves many man-hours of development time by creating program skeletons and avoiding mistakes.



If you are programming with a Crestron framework, the true power of the SIMPL generator can be used to transform your project configuration files into a full, compile-able, program.

Note: This is an advanced programming feature and some cases require existing competency with XML and, if you wish to write your own "Manifest Transformer" plugin, C#.

## A BRIEF OVERVIEW

From the highest level view, the tool takes a collection of data structures that represent a complete Crestron SIMPL Windows program. The tool consumes these data structures and, using official Crestron libraries, builds a SIMPL Windows program. This results in an SMW file, identical to what would be generated by tools such as SIMPL Windows, and can even be compiled by SIMPLified. It is designed to work with complete programs, though the data structures can reflect building blocks, such as hardware devices (e.g. a Touch panel) and logical sub-systems so that large, sophisticated, programs can be built from simpler elements.

Currently we provide two Transformers for the Program Generator. One creates a SIMPL Windows program from XML, the other creates the BACnet structure from csv document. These are explained in more detail later in this document.

The program generator is NOT intended for maintenance of SIMPL programs. This is a one-shot process for creating a program (either complete, or a partial skeleton) and once generated, the standard SIMPL Windows tools should be used to make modifications. Of course, you can always re-generate a program and cut and paste elements from one instance to another.

## DATA STRUCTURES

The main generator data structure is the `Manifest`. This is a C# class which represents a SIMPL program, or one of the building blocks mentioned above. The C# `Manifest` class and its various components are documented in the MSDN-style documentation provided with the package, a link to this can be found under 'Help' in SIMPLified 2.

The classes are deserialisable from an XML document which is defined in the XSD which is hosted by Ultamation. This means that a complete program can be hand-crafted in XML, and processed by the SIMPL generator to create an SMW file (and even a compiled .lpz). At first sight, this may not add much to your workflow productivity, but the real power isn't far away.

The input to the program generator must be a complete `Manifest` object, so – how do you get from an XML document to a finished program?

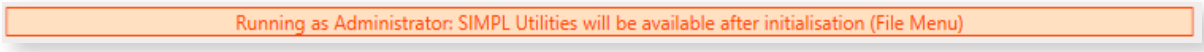
As we mentioned above, the `Manifest` object can be deserialised from XML. This requires some suitable transformation code, and we have provided a plugin in architecture within SIMPLified 2 that allows anyone to develop their own plugin which will consume some input, and generate a Manifest object which is then passed on to the program generator within SIMPLified 2. **We provide some simple Transformers as part of the SIMPLified 2 distribution so that you can start generating programs without any C# knowledge.**

# GETTING STARTED

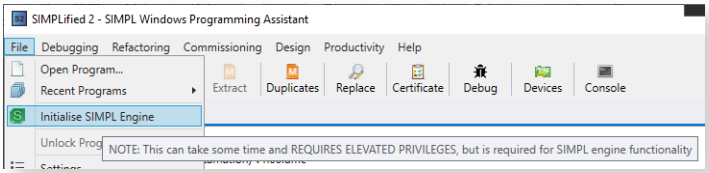
In order to make use of the program generator you must meet the following conditions:

- The latest release of SIMPLified 2
- You must be running SIMPLified 2 as Administrator
  - This is a requirement of the Crestron SIMPL libraries.
- You must have AT LEAST one Transformer plugin.
  - SIMPLified 2 ships with some default transformer plugins (Which are explained later).
  - You can also develop your own Transformer plugin that could convert your own configuration files to our Manifest object to build a system tailored to your own framework. Adding additional transformer plugins is described below.

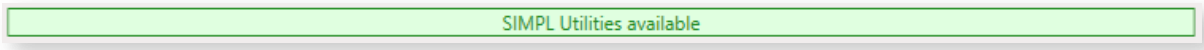
When you run SIMPLified under an administrator account, you will see this notice at the bottom of the program window:




You will need to initialise the native Crestron components that SIMPLified relies on by selecting the menu option shown below:




**Note:** Initialising the SIMPL engine can take a couple of minutes and runs in the UI thread which will make SIMPLified appear to lock up. Please be patient. After successfully initialising the SIMPL engine, this notice will change to:



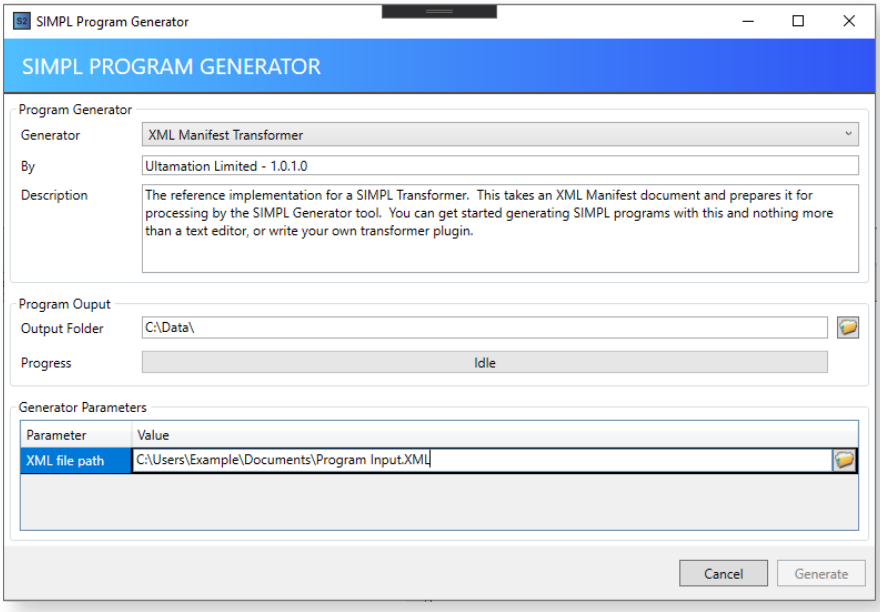
## Installing Additional Transformer Plugins

A transformer plugin is a C# .DLL that implements the ITransformer interface defined in Ultamation's SIMPLManifest assembly, which is provided with SIMPLified 2. Creating a transformer is an advanced topic which is covered later in the document. However, framework developers may provide a plugin for their framework and all you need to do is copy the plugin to the appropriate directory:

- /ProgramData/Ultamation/SIMPLified2/Plugins

Once the file has been copied, restart SIMPLified and if the file has been copied correctly, you will see the plugin appear in the Program Generator transformer drop down.

The default program generator window will look something like this:



## The Default Ultamation Transformers

The easiest way to get started with the program generator is to use the one of the transformers that are provided as part of the installation. We currently provide an XML transformer, and a BACnet Device Tree Builder.

**The XML Transformer**

You will need an XML input file that defines a simple program. The XML document is described and validated by the XML Schema Definition (see below) and this can be used in many XML editors to ensure that your XML is both well-formed (no XML syntax errors) and valid (conforms to the SIMPLified structure).

A program manifest will define at least the program header, the hardware tree and some logic. It can also specify any user module dependencies. When SIMPLified builds a program, it will first copy ALL external dependencies (from a relative file path or user databases) into the build folder to ensure that all dependencies for the target program are satisfied and also so you can ensure you are referencing the correct module versions. As such, ALL user modules for program built with SIMPLified will effectively become project modules, just as they are when you extract a normal program archive.

A program manifest can also reference sub-manifests – these modular XML files that define a specific object (e.g. a TSW) or logic sub-system (e.g. TSW instance code). As a reference is brought in, you can define substitutions that will be applied to signals names, parameters and comments. This provides the mechanism to build programs of any size from previously defined building blocks.

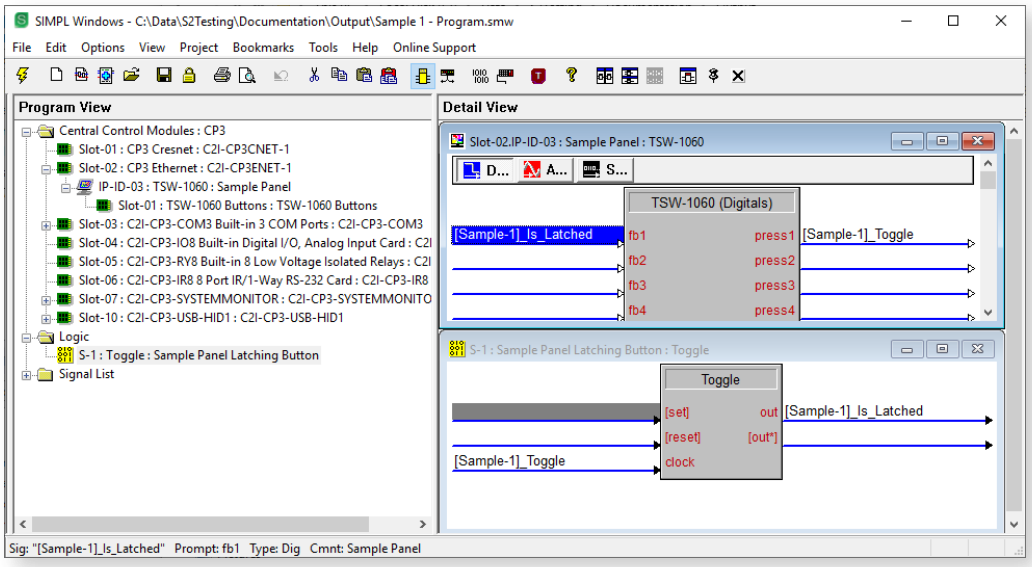
Here is an example of a very simple, but fully-functional, compile-able, program manifest that creates a latching toggle for a touch-panel; a trivial example for a very powerful concept.

All of the examples in this document are available for download from the SIMPLified 2 product page.

```
<?xml version="1.0" encoding="utf-8" ?>


<SimplManifest xmlns = "http://www.ultamation.com"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.ultamation.com
http://research.ultamation.com/schema/2020/SIMPLManifest.xsd">

  <ProgramInfo>
    <ProcessorDeviceName>CP3</ProcessorDeviceName>
    <Programmer>Oliver Hall</Programmer>
    <Dealer>Ultamation Limited</Dealer>
    <ProjectName>Trivial Example</ProjectName>
    <ProjectNo>Sample 1</ProjectNo>
    <ProjectComment>A simple latching button</ProjectComment>
    <ProgramIdTag>Sample1</ProgramIdTag>
  </ProgramInfo>

  <DeviceTree>
    <Ethernet>
      <Device address="03" type="TSW-1060" name="Sample Panel" location="Sampleland">
        <Connections>
            <SignalIn cue="fb1">[Sample-1]_Is_Latched</SignalIn>
            <SignalOut cue="press1">[Sample-1]_Toggle</SignalOut>
        </Connections>
      </Device>
    </Ethernet>
  </DeviceTree>
```

```
  <LogicTree>
    <Symbol>
      <Comment>Sample Panel Latching Button</Comment>
      <Type>Toggle</Type>
      <Connections>
        <SignalIn cue="clock">[Sample-1]_Toggle</SignalIn>
        <SignalOut cue="out">[Sample-1]_Is_Latched</SignalOut>
      </Connections>
    </Symbol>
  </LogicTree>
</SimplManifest>
```

The XML above `Sample 1 - Program.xml` will result in the following program when input to the Program Generator with the XML Transformer:




### *Creating an XML Manifest Input File*

Writing the manifest XML documents out by hand is one option though it is a laborious and error prone exercise. To make things considerably easier, SIMPLified 2 provides a way to export existing program logic and hardware definitions to sub-manifest XML documents. This means you can refine your logic in the usual way (in SIMPL Windows) and once it is proven, you can export a logic tree to build future programs. To do this, open your program in SIMPLified 2, right click on the branch you want to export, and choose 'Export Branch'

There may still be times when it is beneficial to modify the XML files by hand and to aid this manual editing we have provided an XML Schema Definition (.xsd) to provide real-time validation.

We'll now look at the elements of the manifest XML file. For the programmers, you can also refer to the MSDN-style documentation for the `Manifest` class in the namespace `Ultamation.Libs.SimplUtility`.

A manifest document should always start with this document wrapper:

```
<?xml version="1.0" encoding="utf-8" ?>
<SimplManifest xmlns = "http://www.ultamation.com"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.ultamation.com
http://research.ultamation.com/schema/2020/SIMPLManifest.xsd">

<!-- THIS IS WHERE YOU WILL ADD CONTENT -->
```

```
</SimplManifest>
```

This tells the parser that what follows is an XML document, encoded as UTF-8 and the root element is a `SimplManifest` element. This structure is the starting point whether you are creating a program manifest (the main XML document) or a sub-manifest (which will be either a device definition or a program logic sub-system).

### *Program Info (Program Manifest Only)*

You must then define a program info section. This is the only mandatory element for a program and is conceptually equivalent to the Project Program Header in SIMPL Windows. You must specify a target processor. *Warning!* This must EXACTLY match the name you would see in the SIMPL Windows Device Library. Note that some processors have a trailing period (e.g. `PRO3.`)

The `ProgramInfo` structure will look something like this:

```
<ProgramInfo>
  <ProcessorDeviceName>AV3</ProcessorDeviceName>
  <Programmer>Your Name</Programmer>
  <Dealer>Your Company Name</Dealer>
  <ProjectName>Your Project Name</ProjectName>
  <ProjectNo>Your Project Reference</ProjectNo>
  <ProjectComment></ProjectComment>
  <ProgramIdTag>Main</ProgramIdTag>
</ProgramInfo>
```

Most of the data here is informational, with the exception of `ProcessorDeviceName` and `ProgramIdTag` which will have an impact on the executable output.

### *Module Dependencies*

The next section to declare (which isn't shown in the trivial example) is the `Dependencies` section. This is valid for both program manifests and sub-manifests where, for example, a section of logic requires a module.

The example below will pull in a "User Macro" module called "`Power Manager.umc`" which is located in the local machine's "User Macros Directory".

```
<Dependencies>
  <Module>%USERMC%\Power Manager.umc</Module>
</Dependencies>
```

This section should be used to include User Macros (`.umc`) User SIMPL+ modules (`.usp` AND `.ush`) and SIMPL# Libraries (`.clz`). Note – for SIMPL+ modules, you should reference only the `.usp` file (the code) though the `.ush` file (the symbol header) will also be copied. Dependencies are NOT recursive, so if your top level module is itself dependent upon additional modules, you will need to include these other dependencies explicitly.

Module locations can be specified in a number of ways:

- Absolute file paths – are valid, but are usually a bad thing as they are not portable across build environments.
- Relative file paths – are used when you know the location of the dependency relative to the location of the manifest XML file.
- Substitutions – we have defined a number of substitution patterns to allow you to reference modules. These are:
  - %USERDB%\ - The USER IR directory
  - %USERMC%\ - The USER MACROS directory

- %USERSP%\ - The USER SIMPL+/SIMPL# directory

These dependencies will be copied to the output directory prior to program generation.

Note: You DO NOT need to include Crestron Database modules in the `Dependencies` section, though you must ensure that any modules used in the program are present in the Crestron databases on the build machine.

### *Hardware Definitions*

Most programs won't do a great deal without some hardware. The `DeviceTree` section is used to populate the processor's hardware tree, and also defines the signal connections for each device, and any sub-devices/extenders.

Note: At present, SIMPLified only supports Ethernet and Cresnet device trees. We will expand this support to include other hardware elements such as IR drivers and COM ports shortly.

Note: At present, SIMPLified only provides partial support for Smart Object Extenders. We will properly support `.sgd` and CH5 Contract files at the earliest opportunity.

The following example shows a number of hardware elements including a simple touch panel, a reference to a sub-manifest for another touch panel and a remote as a sub-device of a gateway. Each section will be described in turn.

The `DeviceTree` defines a single `Ethernet` and/or `Cresnet` element. Each of these sub-sections defines a collection of `Device` elements.

```
<DeviceTree>
  <Ethernet>
    <Device address="03" type="TSW-1060" name="[TSW1060-1]" location="Sampleland">
      <DeviceCode>5574</DeviceCode>
      <Connections>
        <SignalIn cue="fb1">[TSW1060-1]_Is_Latched</SignalIn>
        <SignalOut cue="press1">[TSW1060-1]_Toggle</SignalOut>
      </Connections>
      <Extenders>
        <Symbol>
          <Type>Activity Detection.</Type>
          <Parameters>
            <Parameter name="Time">60s</Parameter>
          </Parameters>
          <Connections>
            <SignalOut cue="activity">//[TSW1060-1]_Activity</SignalOut>
          </Connections>
        </Symbol>
      </Extenders>
      <SubDevices>
        <Device address="01" type="TSW-1060 Buttons" name="TSW-1060 Buttons" location="Sampleland">
          <Connections>
            <SignalOut cue="Power">[TSW1060-1]_Reset</SignalOut>
            <SignalOut cue="Home">[TSW1060-1]_Set</SignalOut>
          </Connections>
        </Device>
      </SubDevices>
    </Device>
```

The first `Device` is a direct definition of a TSW-1060. The `type` attribute is used to specify the hardware type and, as for the processor in the program info, it must EXACTLY match the name as it appears in the SIMPL Device Library. The `name`, `location` and `ipAddress` attributes are optional.

The program builder will use the type attribute to identify a specific device form the Crestron library, but sometimes there can be variants – such as obsolete versions – which may confuse the program builder. The `DeviceCode` element can be used to explicitly tell the program builder which device you want and this overrides the `type` attribute. The XML export feature described later will always include the `DeviceCode` element to avoid ambiguity.

Each `Device` can specify a collection of signals under the `Connections` element. These are either `SignalIn` or `SignalOut` and you must specify the signal `cue` attribute as it appears on the symbol in SIMPL. The exception to this is for optional cues (denoted by `[…]`) where you do not need to use the enclosing square braces. The signal name can be enclosed in a `<![CDATA[…]]>` section if it requires special characters.

Any parameters can also be specified for a `Device` by using the `Parameters` element. This is similar to the `Connections` collection though each `Parameter` uses the `name` attribute instead of `cue`. An example of this is shown in the `Sample TSW.xml` file.

Devices provide a special collection called `Extenders`. This specifies a collection of device extender symbols. The example above shows the `Activity Detection.` extender and also includes an example of the `Parameter` element. The format of the device extenders is identical to the normal logic tree structure described below though it must naturally follow the permissible objects for the parent device. Any invalid symbols or device extenders will result in a build failure.

Finally, some `Device` elements may also support `SubDevices`. This is a collection of `Device` elements, just like the parent `Device`, so it should be apparent that complex hierarchies can be constructed in this way. Some devices, such as touch panels or keypads include sub-devices for things like physical buttons. Another example of `SubDevices` use is given below in the gateway example where the associated remotes are defined as `SubDevices`.

The next example shows a second touch panel, with identical functionality as the direct definition above, but this time pulled in as a sub-manifest reference, this sub-manifest is contained in the `Sample TSW.xml` file.

```
      <Reference address="04" name="[TSW1060-2]" location="Sampleland Reference">
        <FilePath><![CDATA[Sample TSW.xml]]></FilePath>
        <Substitutions>
          <Definition>
            <Pattern>DeviceName</Pattern>
            <Value>TSW1060-2</Value>
          </Definition>
        </Substitutions>
      </Reference>
```

The sub-manifest file name is relative to the parent XML so, in this case, is in the same directory as the `Sample 2 - Program.xml`. To be useful, sub-manifests will normally include some signal, parameter or comment substitutions. These are defined through the `Substitutions` collection and each `Definition` provides a `Pattern` which will match any signal name, parameter value or comment with the form `%pattern%` and replace that part of the source value with the given `Value`.

In this example, any SIGNAL NAME, PARAMETER or COMMENT that contains the string `%DeviceName%` will have that section replaced by `TSW1060-2`. Therefore, a signal called `[%DeviceName%]_Is_Latched` will become `[TSW1060-2]_Is_Latched`.

A device sub-manifest is very similar to a normal program manifest except it does not define the program info or logic elements and should only define a SINGLE parent device.

The final section shows the `SubDevices` collection being used to define an HR-310 on a gateway at a specific RF-ID. As in the touch panel example, each HR-310 could be defined by a reference to a sub-manifest for the HR-310 detail.

```
    <Device address="0A" type="CEN-GWEXER" name="EX Gateway">
      <SubDevices>
        <Device address="03" type="HR-310" name="[HR310-1]" location="Theatre">
          <Connections>
            <SignalOut cue="Power">[HR310-1]_Reset</SignalOut>
            <SignalOut cue="Menu">[HR310-1]_Set</SignalOut>
          </Connections>
        </Device>
      </SubDevices>
    </Device>
  </Ethernet>
</DeviceTree>
```

### *Program Logic*

Defining the program logic is somewhat similar to the device tree in that the LogicTree element contains a collection of `Symbol`, `SubSystem` or `Reference` elements as show below.

```
<LogicTree>
  <Symbol>
    <Comment>HR310 Latching</Comment>
    <Type>SR</Type>
    <Connections>
      <SignalIn cue="set">[HR310-1]_Set</SignalIn>
      <SignalIn cue="reset">[HR310-1]_Reset</SignalIn>
    </Connections>
  </Symbol>

  <SubSystem>
    <Comment>Touch panel toggles</Comment>
    <Children>
      <Reference>
        <FilePath><![CDATA[Sample Logic.xml]]></FilePath>
        <Substitutions>
          <Definition>
            <Pattern>DeviceName</Pattern>
            <Value>TSW1060-1</Value>
          </Definition>
        </Substitutions>
      </Reference>
      <Reference>
        <FilePath><![CDATA[Sample Logic.xml]]></FilePath>
        <Substitutions>
          <Definition>
            <Pattern>DeviceName</Pattern>
            <Value>TSW1060-2</Value>
          </Definition>
        </Substitutions>
      </Reference>
    </Children>
  </SubSystem>
</LogicTree>
```

This first `Symbol` directly specifies a logic block – the SR latch. The `Type` element is used to define the symbol and you can use the long name of the symbol (e.g. `Set/Reset Latch`) or one of the SIMPL speed key abbreviations (e.g. `SR`) or, in the case of a module reference, the module name including the extension (e.g. `Power Manager.umc`).

The `Connections` element follows the same format as for the `Device` elements. This is also true for `Parameters`. Parameter values (and less usefully input/output cues with default signals) will be populated with their defaults if they are omitted.

The `SubSystem` element is used to define a Subsystem folder in SIMPL. It's only valid child elements are `Comment` and `Children`. `Children` is a collection of logic symbols, just like `LogicTree`.

A `Reference` element is used to bring in a logic sub-manifest, just as the device reference pulls in a device definition with substitutions. Logic sub-manifests can be complex and reference further sub-manifests.

The simple example above pulls in the toggle symbol, similar to that from first, trivial, example but this time we have a substitution defined for DeviceName.

```
<?xml version="1.0" encoding="utf-8" ?>
<SimplManifest xmlns = "http://www.ultamation.com"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.ultamation.com
http://research.ultamation.com/schema/2020/SIMPLManifest.xsd">
  <LogicTree>
     <Symbol>
      <Comment>[%DeviceName%] Logic</Comment>
      <Type>Toggle</Type>
      <Connections>
        <SignalIn cue="set">[%DeviceName%]_Set</SignalIn>
        <SignalIn cue="reset">[%DeviceName%]_Reset</SignalIn>
        <SignalIn cue="clock">[%DeviceName%]_Toggle</SignalIn>
        <SignalOut cue="out">[%DeviceName%]_Is_Latched</SignalOut>
      </Connections>
    </Symbol>
  </LogicTree>
</SimplManifest>
```

It should be apparent from the example above that large, sophisticated, and complete programs can be generated from these XML snippets. The `Sample 2 – Program.xml` and associated sub-manifest files does, in fact, generate a complete program with zero compiler notices or warnings.

### *A Note on Symbols*

The build process will always attempt to "complete" a symbol. This means that if you omit a signal for a cue or parameter value, the build process will apply the default signal, parameter value or – in the case of empty cues, apply the special '//' signal.

There are still ways in which a program can be generated that will either not compile at all, or compile with errors, notices and warnings – it is still your responsibility to define the program correctly. SIMPLified's static analysis features can help identify issues.

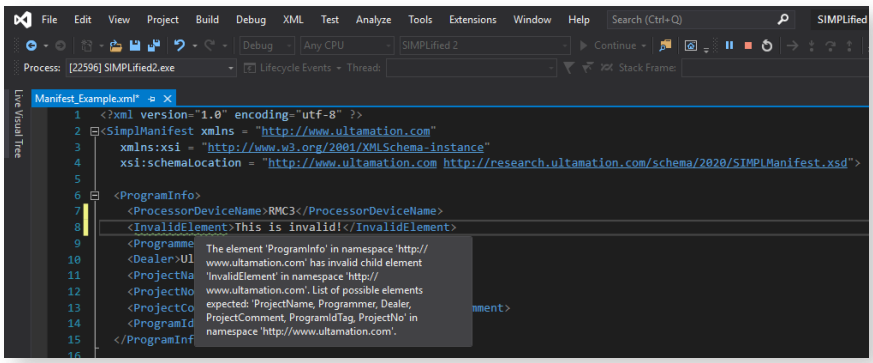
### *Using the XSD to Validate your XML*

Many text editors assist with creating XML documents though syntax colour, auto-completion and validation. One technique for this uses an XML Schema Definition which is itself an XML document that defines the valid structure of the document you will be editing.

Your XML manifest files should start with the following declarations to specify where the XSD file is located:

```
<SimplManifest xmlns = "http://www.ultamation.com"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.ultamation.com
http://research.ultamation.com/schema/2020/SIMPLManifest.xsd">
```

This tells the XML consumers the location of the `SIMPLManifest.xsd` which Ultamation will maintain. The example below shows Visual Studio notifying us that the XML is invalid because a foreign element (`<InvalidElement>`) has been introduced in the wrong place.




Using the XSD is not a requirement, but we strongly suggest you validate your XML if you experience any issues as even well-formed XML (where the syntax is correct) can fail to produce expected program output.

A "well-formed", yet "invalid", example:

```
    <SubSystem>
      <Comment>Touch panel toggles</Comment>
      <!-- <Children> -->
        <Reference>
          <FilePath><![CDATA[Sample Logic.xml]]></FilePath>
          <Substitutions>
            <Definition>
              <Pattern>DeviceName</Pattern>
              <Value>TSW1060-1</Value>
            </Definition>
          </Substitutions>
        </Reference>
      <!-- </Children> -->
    </SubSystem>
```

The example above is perfectly valid XML, but we have commented out the Children element. The transformer plug will read this XML, and ignore the `<Reference>` element as it would only expect this element to be declared within a `<Children>` element collection. This will result in an empty Subsystem element.

### *Creating Reference Sub-Manifest Input Files*

A complete program, defined in a single XML file, could be a huge document and provide very little benefit over creating the program in SIMPL Windows. As we have already alluded to in the earlier examples, we have provided a way to reference re-usable device and logic manifest sub-files which can be pulled into the main manifest, and substitutions can be applied in the process to modify signal names, comments and parameters.

```
<Reference>
  <FilePath><![CDATA[Sample Logic.xml]]></FilePath>
  <Substitutions>
    <Definition>
      <Pattern>DeviceName</Pattern>
      <Value>TSW1060-2</Value>
    </Definition>
  </Substitutions>
</Reference>
```

A `Reference` is defined where a `Device` or `Symbol` element would normally be placed, and it must include a child `FilePath` element which points to the sub-manifest file. This file path can be absolute though it is more portable to use relative references. Each relative path is relative to the containing XML file.

Hint: The example above shows the file path wrapped in a CDATA section though this is not mandatory.

For `Device` `Reference` elements, you can also specify a number of optional attributes which will override the attributes specified on the root element of the referenced sub-manifest. These are:

- address – the hexadecimal address for the referenced device. This might be the IP-ID, Cresnet ID, RF-ID or – more generally – the hardware "slot" number. Unlike SIMPL, which will find an appropriate location when devices are added to the device tree, addresses MUST be explicitly stated, and it is your responsibility to specify only addresses that are valid. The program builder will generate an error or unexpected output if you attempt to add a device to an invalid slot or on-top of a device that already occupies the specified slot.
- name – define the Name meta-data for the device.
- location – define the Location meta-data for the device.

These attributes are ignored for logic sub-manifest references.

The flexibility of sub-manifests comes from being able to make signal, parameter and comment substitutions during the import. These are defined in a `Substitutions` element collection and each substitution is described by a `Definition` element which has two children – the `Pattern` element which defines the string to match and the `Value` element which defines what each instance of the pattern will be substituted with.

You can specify as many substitutions definitions as you wish, though logical structures that follow well-encapsulated design should not become unwieldy. If you find you need to define a large number of signal substitutions, this may suggest a "code smell[1]".

[1] A code smell is an element of code or system design that doesn't feel or "smell" right. An over-abundance of code smells should lead you towards re-design or refactoring in order to reduce technical debt.

### *The Sub-Manifest*

The sub-manifest XML file has essentially the same structure as the main program manifest though with some notable exceptions.

As previously noted, a sub-manifest will not include a `ProgramInfo` element should only be used to define a single device.

We will revisit the earlier example; this time looking at the `Sample TSW.xml` file which was used to define one of the TSW-1060 touch panels.

The device will take default values for address, name and location though these would be overridden by the Reference attributes.

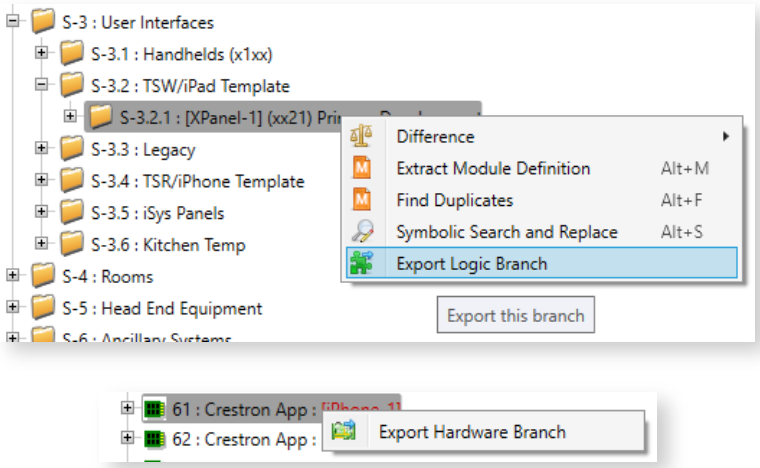
The `Device` defines parameters, a device extender (with a further parameter) and a sub device for the buttons. This sub-device would, in fact, be added automatically if this section were to be omitted, but it is required here so that we can assign some signals to the hard buttons.

Where you will make substitutions in the sub-manifest, you should place the `Pattern` that will be defined in the substitution definition of the reference element and enclose it with `%...%`. The pattern is also case sensitive.

```
<?xml version="1.0" encoding="utf-8" ?>
<SimplManifest xmlns = "http://www.ultamation.com"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.ultamation.com
http://research.ultamation.com/schema/2020/SIMPLManifest.xsd">
  <DeviceTree>
    <Ethernet>
      <Device address="03" type="TSW-1060" name="Sample Panel" location="Sampleland">
        <Parameters>
          <Parameter name="PermanentStringSize">255d</Parameter>
        </Parameters>
        <Connections>
          <SignalIn cue="fb1">[%DeviceName%]_Is_Latched</SignalIn>
          <SignalOut cue="press1">[%DeviceName%]_Toggle</SignalOut>
        </Connections>
        <Extenders>
          <Symbol>
            <Type>Activity Detection.</Type>
            <Parameters>
              <Parameter name="Time">60s</Parameter>
            </Parameters>
            <Connections>
              <SignalOut cue="activity">//[%DeviceName%]_Activity</SignalOut>
            </Connections>
          </Symbol>
        </Extenders>
        <SubDevices>
          <Device address="01" type="TSW-1060 Buttons" name="TSW-1060 Buttons" location="Sampleland">
            <Connections>
              <SignalOut cue="Power">[%DeviceName%]_Reset</SignalOut>
              <SignalOut cue="Home">[%DeviceName%]_Set</SignalOut>
            </Connections>
          </Device>
        </SubDevices>
      </Device>
    </Ethernet>
  </DeviceTree>
</SimplManifest>
```

### *Exporting Existing Logic and Devices to Sub-Manifest Input Files*

Hand crafting large XML files can be a laborious process, even if you use an XML tool to assist and validate your document. We have provided a mechanism within SIMPLified 2 to generate XML sub-manifest files from an existing program. This way, you can take existing, proven logic, export an instance of a device or logic sub-system to a sub-manifest file and then use this as the basis for references in new programs.

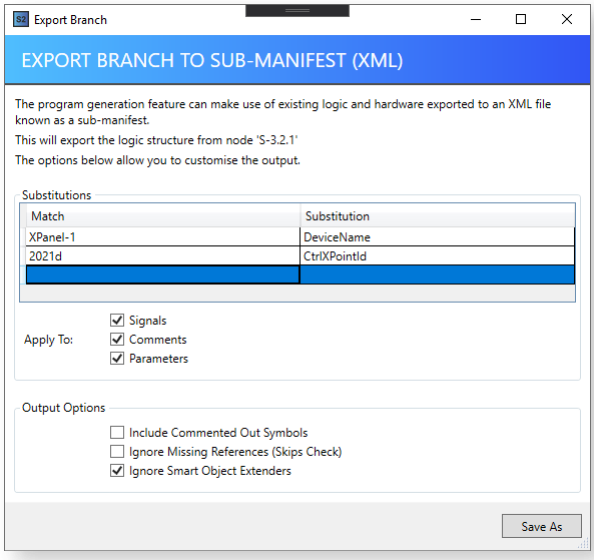






Open either the Logic Tree or the Device Tree for an existing program and open the context menu. From here you will be able to Export a Logic Branch or Hardware Branch.

This will then open the dialog below:




By declaring substitutions we wish to make during export, we can potentially create the sub-manifest without any further user modifications. In the example above, any occurrences of XPanel-1 will be replaced by the substitution pattern `%DeviceName%`.

Note: you DO NOT include the special pattern delimiters `%...%` in the Substitution value for the export.

This obviously requires care, and considered use of signal naming to ensure that you only target the signals, parameters or comments that you wish to manipulate. To limit the pattern substitution scope, you can elect for this to apply only to specific types, though these settings are global.

Automatic substitutions are not to be considered a complete solution to creating sub-manifests and for more complex logic or devices, it is quite likely that you will need to perform some manual editing of the exported output to tailor the sub-manifest for re-import.

There are some further options for controlling the output for the export.

- By default, an export will ignore commented sections of the logic or device tree. If you wish to include these in the export, select the "Include Commented-Out Symbols" check box. Bear in mind though; anything in the export will be created in the generated program – NOT commented out.
- You can skip any module reference checks if this would otherwise prevent the export from completing. This can happen if you don't have access to a required dependency yet you still wish to create the export and manually edit the output. Missing dependencies will result in incorrect signal cues and parameter names in the export.
- At the present time, SIMPLified 2 is unable to FULLY support Smart Graphics Extenders. This is due to limitations in the Crestron libraries and CED support in SIMPL Windows and we will resolve this as soon as we are able. As a result of this, the default behaviour is to ignore Smart Object Extenders. SIMPLified 2 WILL export the extenders correctly, but the build process currently requires the `.CED` files rather than the newer `.SGD` files and some .CEDs are not correctly supported in SIMPL Windows.

When you select Save As, specify the output filename for the sub-manifest. As well as generating an XML file for the selected branch, the export process will also resolve any module dependencies and make a copy of each dependency into a sub-directory in the same folder as the export file.

The export file will include a dependencies section for these modules with the correct relative path. You should therefore take care in moving dependencies relative to the XML export file.

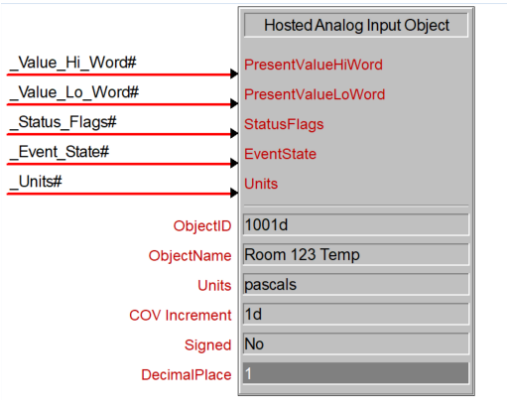
**BACnet Device Tree Transformer**

The second transformer included with Simplified 2 is the BACnet Device Tree Builder. Instead of an XML document like the first transformer, this requires a csv file to define the BACnet structure and devices. This is our example document `BACnet Input Examplpe.csv` downloadable from the SIMPLified 2 shop page.

This is a basic example, for a more detailed use case, refer to the `SIMPLified 2 Bacnet Worked Example` document, downloadable on the shop page or accessible by the SIMPLified 2 help menu.

| Omit | DataType | Name | Id | Instance | PortNumber | Covinc | Sign | Polarity | States | CovSub | Scan | Priority | CovLife | DeviceManifest | LogicManifest | Units | Decimals | Substitutions |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Hosted | | 101 | | | | | | | | | | | | | | | |
| | AI | Room 123 Temp | 1001 | | | | | | | | | | | | BACnet_Analog_Input.xml | 53 | 1 | |
| y | AV | Room 123 SetPoint | 1002 | | | | | | | | | | | | RoomMode.xml | 47 | | |
| | BI | Room 123 Demand | 1003 | | | | | | | | | | | | | | | |
| y | BV | Room 123 Mode | 1004 | | | | | | | | | | | | BACnet_Binary_Value.xml | | | |
| y | MI | Room 456 Wibble | 1005 | | | | | | | | | | | | BACnet_Multistate_Input.xml | | | |
| y | MV | Room 456 Wobble | 1006 | | | | | | | | | | | | | | | |
| | Remote | | 301 | | | | | | | | | | | | | | | |
| | BI | | | | | | | | | | | | | | | | | |
| | Remote | | 201 | | | | | | | | | | | | | | | |
| | AI | Remote Analog Input | 2001 | | | | | | | | | | | | | | | |

When this file is given to the program generator with the BACnet Device Tree Transformer selected, it will output an smw file with the following BACnet structure. And you can see all the fields have been populated with the information provided in the csv file. This is the Hosted Analog Input Object from row three of the example file:


### *Creating the BACnet csv File*

You can use the example document as a template for your own BACnet devices. Simply fill in the device information in each row. If you are using a spreadsheet like the example, the document should be saved as a csv file before using in the program generator.

To add *hosted devices*, create one row with 'Hosted' as the DataType, then list all the hosted devices under it, using one row per device. **Note:** There should only be one Hosted section in the file.

To add *remote hosted devices*, add a row with 'Remote' as the DataType, then list the remote devices underneath using one row for each device. You can have multiple remote host sections.

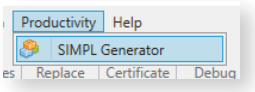
### Columns Explained

Listed below are the column headers for the BACnet Transformer input file with a description for each

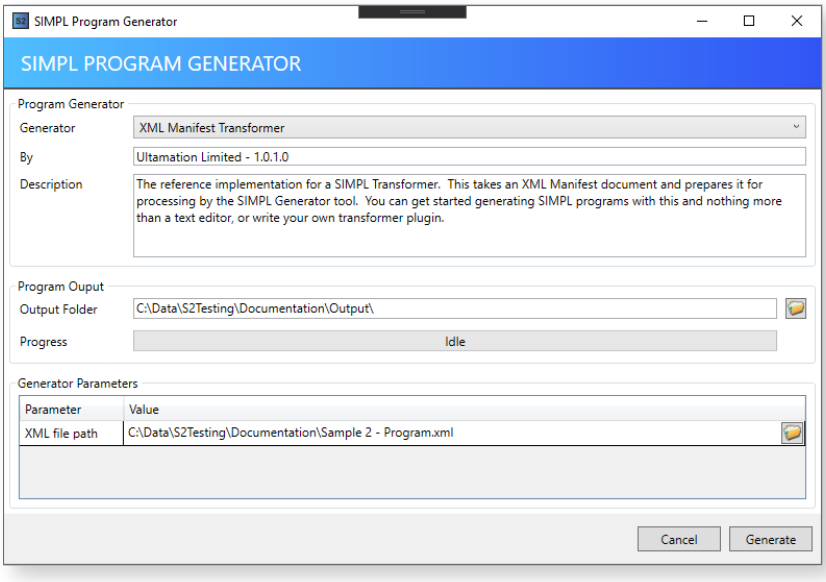
| Column Header | Description | Value Type |
|---|---|---|
| Omit | Type anything into this column if you want to omit the device in this row from being included in the generated program. If the field is empty then the device will be included. **Note:** if a Hosted or Remote row is labelled to be omitted, all sub devices will also be omitted. | Any |
| DataType | To begin a Hosted Devices section, write 'Hosted' in this field, then the rows beneath will be the sub devices. To start a Remote Hosted Devices section, write 'Remote' here with remote devices in the rows beneath. For a device row, this should be the device's abbreviation, e.g. a Analog Input Object would be 'AI' in this field. A Multistate Output Object would be 'MO'. | String |
| Name | The object name parameter in the symbol. | String |
| Id | The symbol's object ID parameter value. | Int |
| Instance | Defines the value that will be substituted for "bacnetDevice" in the signal and logic definitions | String |
| PortNumber | The symbol's port parameter value. | Int |
| Covinc | The symbol's COV Increment parameter value. | Int |
| Sign | Whether the device is signed or unsigned. | Yes or no |
| Polarity | The symbol's polarity parameter value. | |
| States | The symbol's Number of States parameter | Int |
| CovSub | The symbol's COV Subscribe parameter | Yes or no |
| Scan | The symbol's Scan Rate parameter in seconds | Int |
| Priority | The symbol's Priority parameter | Int |
| CovLife | The symbol's COV Lifetime parameter | Int |
| DeviceManifest | Specifies that an external Manifest file is to be loaded and used to define the signal connections for the hardware object and any default parameter values. These values can then be overridden from the CSV file if they are defined (e.g. "Decimals"in this example.) | File name |
| LogicManifest | Specifies a single logic folder. | File name |
| Units | The symbol's Units parameter | Int |
| Decimals | The symbol's DecimalPlace parameter | Int |
| Substitutions | Provides a way for the CSV file to override signal, comment and parameter values with row specific values. Name value substitution pairs are defined by NAME=VALUE and multiple values are separated using the pipe '|'symbol | String |

# RUNNING THE PROGRAM GENERATOR

Once you have suitable input files prepared and validated, you are ready to generate a SIMPL program. This facility is enabled when you have an active Productivity Entitlement.




The SIMPL Generator will open the program generation window. From here you will select the appropriate generator plugin. The default plugin for XML manifests is shown below.




Regardless of which transformer you use, you must always specify an output folder which is where the generated program and any dependencies will be placed (and compiled, if you elect to do so). The folder icon to the right allows you to browse to a destination folder and optionally create new sub-folders.

Based on the selected generator plugin, the lower part of the dialog will reflect any parameters that can be specified. In the case of the default XML transformer plugin, the only parameter is the input XML filename. Other plugins can specify any parameters they wish, and these could be file paths to configuration files, database credentials, tokens, etc.

The generator parameters are defined by the plugin, and it is the responsibility of the plugin to persist (if required) and validate the values.

The generate button is blocked until all of the parameters required by the plugin are satisfied, When you hit Generate, the following things happen:

- The selected plugin's `GenerateManifest(…)` method is called with the supplied parameters. This calls the plugin's custom implementation and the result, if successful, is a complete program `Manifest` structure (a C# object this time, as opposed to an XML document).
- The `Manifest` object is passed to SIMPLified's program builder which performs the program generation. If the program generation finds any errors, these are reported back to the user so that corrections can be made.
- When the build process is completed without errors, SIMPLified then calls the plugin's `OnSuccessfulCompletion()` method. This offers the plugin an opportunity to make final checks (such as validating an access token supplied in the generator arguments) before writing the generated program file to disk.

- Once the program `.smw` has been written to disk, SIMPLified will offer you the option of compiling the program. You can say **Yes** to launch the SIMPL compiler outside of SIMPL Windows and continue to work on other programming tasks, or say **No** and load the generated `.smw` into SIMPL to make adjustments before final compilation within SIMPL Windows.

Of course, you can also load the `.smw` into SIMPLified to check for potential issues through the static analysis functions.

You're now ready to start generating complete SIMPL Windows programs from nothing more than an XML document. Enjoy!

### Or you could take things further:

If you work with a framework that has a common logical structure and where system components can be inferred or specified through a configuration file or database, then you can implement a plugin for your own use, or for customers that use your framework.

A custom plugin can take input from any source, and programmatically generate the `Manifest` object based on your framework's operational/business logic. This can still reference sub-manifest XML files so you can exploit the power of program generated structures with exported snippets to complete programs of any complexity – and still retain the facility to modify the final program output within SIMPL Windows to offer your clients the famous Crestron customisation.

# CREATING CUSTOM TRANSFORMER PLUGINS

The following sections require experience with C# programming. The objects described in the following section are all defined in the namespace `Ultamation.Libs.SimplUtility`. The documentation for this namespace can be found at:

`http://research.ultamation.com/reference/api/simpl`

SIMPLified2's Program Generation feature is centred around a plugin architecture known as a "Transformer" object. Every transformer must implement the following interface:

```
Ultamation.Libs.SimplUtility.ITransformer
```

The job of a transformer is to take some input, and transform it into a `Manifest` object that SIMPLified's program generator can use.

A transformer will define a number of meta-data properties, such as `FriendlyName`, `Description`, `Publisher`, a collection of generator parameters (`TransformerArg`), and the methods required by SIMPLified: `GenerateManifest(…)` and `OnSuccessfulCompletion()`.

Create a standard .NET Framework Class Library project, add a reference to the `SIMPLManifest` assembly provided with SIMPLified and write a class that implements `ITransformer`. SIMPLified will present your `TransformerArg` objects to the user via the program generator dialog and then call your `GenerateManifest(…)` method with the user supplied `TransformerArg` object values. Your method will then read the specified input, build a Manifest object and return it to SIMPLified. SIMPLified will then call `OnSuccessfulCompletion()` so that you can make any final checks, or simply return True to save the generated program.

We have provided the default XML Transformer plugin as a working example, though the actual program generation is very simple in this case – being left to the XML deserialiser. We will provide further examples over time.