AllJoyn XML to C++ code generator

Requirements

80‑xxxxx‑x Rev. x

August 25, 2011

Submit technical questions at:

<https://www.alljoyn.org/forums>

Revision history

|  |  |  |
| --- | --- | --- |
| Revision | Date | Description |
| A | June 2010 | Initial release |
| B | August 25, 2011 | Updated usage. Changed quicBus to AllJoyn. |

Contents

No table of contents entries found.

Figures

**No table of figures entries found.**

Tables

**No table of figures entries found.**

# Introduction

The AllJoyn XML to C++ code generator tool is intended to convert an XML description of an AllJoyn Service Object (the same that would be gotten via Introspection) to C++ template code that will allow developers to easily create AllJoyn Services and Clients. This document’s purpose is to define the usage of the tool and its requirements.

# 

# Tool Usage

ajgen –h | --help

Display this help

ajgen --well-known-name | -w <well\_known\_name>

[–l] [--object-name | -n <object\_name>] [--root-path | -r <obj\_path>]

[--client-only | -c] [-empty-elements | -e]

[--output-path | -p <path>] [--overwrite | -o]

[--user-output | -u {fweid}]

[--runnable | -R]

[xml\_file]+

-w: the well-known name that the object will use when requesting a bus

name or advertising a name.

-l: relaxes the requirement that all method and signal arguments to be

named. If this is set default names will be generated for

arguments without names.

-b: The object path (including name) of the base node defined in the

xml\_file(s). See the Naming Rules section below for usage.

-c: only generate the client side code; without this flag both the

client and service code are generated.

-e: Will allow empty elements (i.e. objects and interfaces) to be

defined in the xml\_file(s). It this is not set empty elements will

cause the tool will exit with an error.

-p: the path where the generated C++ files will be placed. If not

included they will be output in the current working directory.

-o: overwrite files if they exist in the output path. If this flag is

not set, then exiting files will not be overwritten, and the tool

will exit with an error.

-u: sets the logging level to

f – fatal

w – warning

e – error

i – inform

d – detail

-R: the generated client will make method calls with default values

and the service method handlers will reply with default values.

This option requires a valid object path to be specified (i.e. –r).

xml\_file: one or more files containing the xml definition of an

object’s interface(s).

## Future flags

-nei: disallow empty interfaces

-neo: disallow empty objects

# Requirements

## The executable

1. The tool must be a stand alone executable with versions that will run under Windows, and Linux; specifically, it should not be necessary to invoke the Java interpreter to run the tool.
2. The tool must support the usage as defined in section 2.

## General Output

1. The tool must include the license terms in the code for files that are not explicitly generated – see requirements 6 and 7.
   1. This shall be done in the block comments at the top of each file generated.
2. The tool must generate **all** of the code required to create both a service and a client
   1. this includes all of the code that is used to connect & register with the bus
3. The tool will also generate sample “main” files that contain all of the calls to create the service and sample client defined in the XML file(s).
   1. In the case of the service the file will be structured such that it assumes no other service is going to be supported in the same executable – however will be commented to make it obvious how to support multiple services if that is desired.
   2. In the case of the client file a comment will indicate where the invocation of the service code should happen.
4. The distribution will include a BusManager class files for which will be copied from the distribution directory to the output directory.
5. The distribution will include instructions on how to build and link: specifically instructions on what needs to be added to the header and library search paths to successfully build.
   1. These instructions will be copied in the same way as the bus manager files.
6. All of the generated code will be well commented such that it is clear to the developer how the code works – this is especially true of the “boilerplate” code.
7. All of the code that developers need to modify/add to complete the implementation shall be in a single file for the service and client (file details are described in requirement 10 below).
   1. These files will only contain the skeletons of the methods that need implementing
   2. The skeleton methods will have the same signatures that the XML file describes for the methods and signals.
   3. The rest of the auto-generated “boilerplate” code will be in other files
8. The output files shall be the following:

|  |  |
| --- | --- |
| **Service** | |
| <object\_name>ServiceMain.cc | Sample file that contains calls to attach the application to the bus, and expose the service object. This is a sample file as it is possible that one application may wish to support multiple service objects. |
| <object\_name>Service.h | The header file that defines *all* of the methods that are used by the service object. This includes the “boilerplate” methods as well as the methods that the developer will actually have to provide implementations for. It also includes any type definitions for compound types that are part of the method definitions. |
| <object\_name>Service.cc | Service object “boilerplate” code that implements all of the service functionality that *does not* need to be modified by the developer. This includes generating the messages that correspond to the various methods and signals that comprise the service object. |
| <object\_name>ServiceMethods.cc | Skeleton code for the methods that the developer needs to implement. The methods in here will look like the XML descriptions in the input files. |
| **Client** | |
| <object\_name>ClientMain.cc | Sample file that contains calls to attach the application to the bus. This is a sample file to help the application writer understand what needs to be done in order to interact with the service objects on the bus. |
| <object\_name>Client.h | The header file that defines *all* of the methods that are used by the client to interact with the service object |
| <object\_name>Client.cc | Client “boilerplate” code that implements the required client functionality that does not need to be modified by the developer. This is mainly the code that wraps the marshaling and un-marshaling of the messages, as well as the bus interaction for methods, signals and properties. |
| <object\_name>ClientHandlers.cc | Skeleton code for the signal handling that the developer needs to implement. Depending on asynchronous method handling there may also be handlers for method replies here too. |
| **Utility** | |
| BusAttachmentMgr.h | Header file for the bus attachment manager. |
| BusAttachmentMgr.cc | The bus attachment manager: a class that simplifies the interaction with the bus. |

## Code Structure

1. The tool shall support the generation of both the client and service method calls.
   1. On the client side this method call will marshal the ‘in’ parameters to the method call into a quicBus message.
   2. On the service side this will be skeleton code that the developer then uses to implement the actual functionality associated with the service. There will be two parts to this code:
      1. The developer defined method (DDM) – this is the code that the developer wrote to implement the skeleton the tool generated.
      2. The “wrapper” code that will deal with unmarshaling the message and will call the DDM. If the method is synchronous it will call the method that will marshal the ‘out’ parameters and send the reply message.
   3. A mechanism for supporting asynchronous method calls will be supported by having a method that is used for marshalling the ‘out’ parameters into a reply message and sending it.

## XML Parsing

1. The tool shall support the generation of properties
2. The tool shall support object hierarchies (i.e. it will support nested objects)
3. The tool shall deal with complex data types:
   1. dictionary entries
   2. variant types
   3. nested structures ( max specified depth is 32)
   4. nested arrays ( max specified depth is 32)
4. Any compound data type shall be named for the argument/property that it is defined for.
5. The tool will ensure that there is a name for the XML object being defined (this is done either in the XML or via the appropriate command line flag ().
   1. This name will be used to name the C++ class that will represent the service.
   2. The naming rules (i.e. how the name is specified) depend on the XML structure see section 3.1.1 below.
6. The tool will support annotations; see section 3.4.2 below.
7. The tool will generate either an error or an empty object or interface (see discussion in section 3.4.3.2 below).
8. The tool will generate an error when it encounters an annotation that it doesn’t explicitly support (see discussion in section 3.4.3.2 below)

### Naming Rules

#### Class Names

Classes are named by the interfaces that they are comprised of. At first, classes will be named using only the last section (after the last ‘.’) of the interface names. However, if there is a name collision, then the full interface names will be used to generate the class names. If there is even one class name collision , then all classes in the entire xml file will be generated using full interface names, even if they weren’t one of the colliding classes.

Given a node that contains an interface named com.qcom.A then the class generated will be called **A** and the node is an instance of class **A**. If a node contains multiple interfaces, then the full interface names will be sorted in alphabetical order and, assuming no naming collisions, the last sections will be combined to form the class name. For example, a node containing interfaces com.qcom.A and com.qcom.B will be an instance of a class called **A\_\_B**.

If there is a name conflict between class names, then the entire interface name is used as the class name. Given a node that implements an interface com.qcom.A and another node that implements an interface com.quic.A, there will be two classes generated, **com\_qcom\_A** and **com\_quic\_A**. Similarly, if there was a class name conflict between classes that comprise of multiple interfaces, one class having interfaces com.qcom.A and com.qcom.B and another having interfaces com.quic.A and com.quic.B, the resulting class names would be **com\_qcom\_A\_\_com\_qcom\_B** and **com\_quic\_A\_\_com\_quic\_B**.

#### Instance Names

The name attributes of the node tags in the xml files specify the instance names. If there is an instance name conflict then the node’s full object path becomes the instance name instead. If there is even one instance name conflict, then all instance names in the entire xml file are updated to use their full object paths as names.

This section describes the rules around how names of services (or the quicBus objects that represent them) are given to the code generator. There are two ways names can be specififed:

* On the command line (-b)
* In the XML file

Each has specific rules that define how these will work.

First (and most simply) the command line option (see –b in section 2 above). The command line option can either be given an Instance Object Path (IOP) or a Set Object Path (SOP). This option assigns the name attribute of only the **base** node.

In the XML file, the name attribute of the base node can be either be an IOP, a SOP or just an instance name. Nested nodes **must** always have name attributes without exception. The names of nested nodes cannot contain object paths. They must only be instance names.

Here are the rules defining the valid interaction between the xml file and the –b command line option:

If there is no name attribute specified by the xml file for the base node and the –b command line option is not used, then the test xml file is only valid if the base node has named children nodes. The base node will be assumed to be the root quicBus node of the object hierarchy:‘/’, and no explicit code will be generated for this node (as the quicBus system will automatically generate that node). Otherwise an error will be thrown.

|  |  |  |
| --- | --- | --- |
| -b value | Acceptable base node XML value | Example |
| None | Unnamed IFF base node has no interfaces and contains named child node | See above  -b:  XML: |
| None | Instance Name only | -b:  XML: name=“foo” |
| None | Instance Object Path (IOP) | -b:  XML: name=/foo/bar |
| None | Set Object Path (SOP) IFF base node has no interfaces and contains named child node | -b:  XML: name=”/foo/” |
| Set Object Path (SOP) – has ‘/’ at start and end | Base node contains the Instance Name only (contains no ‘/’ characters) | -b: /foo/  XML: name=”bar” |
| SOP | Unnamed IFF base node has no interfaces and contains named child node | -b:/foo/  XML: |
| SOP | Exact Match IFF base node has no interfaces and contains named child node | -b:/foo/  XML:name=”/foo/” |
| SOP | IOP IFF IOP = SOP (from –b) + Instance Name | -b: /a/  XML: name=”/a/b” |
| Instance Object Path (IOP) – does not have a trailing ‘/’ | Unnamed | -b: /foo/bar  XML: |
| IOP | Exact match | -b: /foo/bar  XML: name=”/foo/bar” |
| IOP | Instance Name matches | -b: /foo/bar  XML: name=”bar” |
| IOP | SOP IFF SOP = IOP (from –b) – Instance Name | -b:/foo/bar  XML:/foo/ |

### XML Annotations

The following annotations will be supported:

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| 1. org.freedesktop.DBus.Method.NoReply | true or false | If set, don't expect a reply to the method call; defaults to false. |
| 1. org.Alljoyn.bus.Item.IsSecure | true or false | Requires the method to be secured using authentication and encryption. Results in a falg being set in the method invocation |
| 1. org.Alljoyn.bus.Arg.VariantTypes | String containing a comma separated list of types. | This provides a hint to a code generator about what the contents of a Variant is expected to be; e.g. “y,b,s” means variant will contain one of an byte, boolean or a string. |

### XML Schema and the DTD

The code generator will use a validating parser, to validate XML input from the user and to identify specific conditions in the XML. Since the DBus DTD is not well implemented, validation will be done using an XML schema.

#### Differences with the DBus DTD

The schema is based on the DBus DTD, but performs validation with more granular control.

Specifically, the schema will have the following differences from the DBus DTD. Unless specifically contradicted by this list, all properties/requirements of the DBus DTD remain effective.

1. No required sequences.  The DTD requires sequenced elements.  For example, ‘interface’ is required to present the following order:  annotation, method, signal, property.  The schema will not have that restriction.
2. The ‘name’ attribute for ‘node’ is only required for nested nodes (i.e. nodes contained inside of other nodes) and not for the parent node (i.e. highest level node).
3. ‘node’ must contain at least one interface or node (see bullet 1 in section 3.4.3.2 below).  The DTD permits an empty node.
4. Interface must contain at least one method, signal, or property(see bullet 1 in section 3.4.3.2 below).
   1. If the interface contains at least one method, signal, or property, then optionally, it may also have one or more annotations.  The DTD permits an empty interface.
5. The direction attribute for ‘arg’ is only valid when used with methods.  The DTD permits the direction attribute for all applications of ‘arg’.
6. The ‘type’ attribute for ‘arg’ and ‘property’ is restricted to a maximum 255 character string that matches a character set (i.e. regular expression).  The DTD permits any string.
7. The ‘name’ attribute for ‘annotation’ will be limited to a list of enumerated values (see bullet 2 in section 3.4.3.2 below).
8. ‘signal’ may have zero or more instances of ‘arg’ and ‘annotation’.  The DTD requires exactly one, each.
9. ‘property’ may have zero or more annotations.  The DTD requires exactly one annotation.

#### Handling exceptions due to schema violation

Violations of the schema by user input are not necessarily prohibited. In some circumstances, they should be used to warn the user that they have provided “unusual” input and that they should ensure that that was their intent.

Specifically, users should be warned (but not prohibited) when generating the following exceptions. All other exceptions should be considered fatal.

1. Input XML contains an empty ‘node’ or empty ‘interface’.
2. The ‘name’ attribute for an ‘annotation’ element is not contained in the enumerated list.

# Limitations/ToDo

## Current Status

Nested nodes (see below) are not supported

## Release 1.0

* 1. Multidimensional arrays are not supported (requirement 14.d)
  2. Annotation defined in section 3.4.2 (requirement 22) is not supported
  3. Containers of variants (requirement 14.b)

Variants will be treated as an unsigned int array. For example a dictionary of variants is treated as just an unsigned int array and not as a Dictionary of unsigned int arrays.

* 1. Arguments and Properties cannot have the name "status"

Method and signal arguments as well as properties cannot be named "status" because that is a local QStatus variable being used by the code.

# Open Issues/Design Decisions

# Glossary

* **Base Node** – the topmost/outermost node
* **Instance Name** – the name of an object instance that must not contain a '/'
* **IOP** – Instance Object Path (has a '/' at the start, but does NOT have a trailing '/').
  + Ex: */com/qcom/Alljoyn*
* **Root Node** – ‘/’
* **SOP** – Set Object Path (has a '/' at the start and end).
  + Ex: */com/qcom/*