Comparison of Franca IDL Mappings

The mapping presented below is based on the following documentation:

- Common API C++ Specification, v2.1.6
- IPC Quartztime, commit 5288d9796132069f2eb39a440ce229f8defe1033 (2015-02-07)
- IPC Quartztime, commit d877e3ca963d775a951f40d804555de868b3052c (version 0.80)

Franca Element	**Common API C++**	**IPC Quartztime**	**Common API C**
Directory hierarchy	- Directory hierarchy of input `.fidl` files does not affect the generated code. (GENIVI rules for Franca IDL usage recommend that `.fidl` files are stored in a directory hierarchy that matches the corresponding package hierarchy.)	- Directory hierarchy of input `.fidl` files does not affect the generated code.	
File name	- Names of input `.fidl` files do not affect the generated code. Code generator processes each `.fidl` file in isolation. - Individual generated files are created at the `interface` and `typeCollection` level.	- Names of input `.fidl` files do not affect the generated code. Code generator processes each `.fidl` file in isolation.	
Generic elements			
Identifiers	Used mostly verbatim: - `attribute`'s and `broadcasts`'s are implemented via classes and methods that incorporate their names (e.g. `<name>Attribute` and `get<name>Attribute()`, where `<name>` is forced to start with an uppercase letter). - `method`'s are implemented in multiple variants that might use decorated names (e.g. `Async` suffix).	Used mostly decorated: - All elements defined inside a `typeCollection` or an `interface` use a prefix that includes their container name `"ipc_<name>_*"`.	
Comments	- Structured comments (`<<** ... **>>`) are translated into Doxygen comments for the corresponding element. - Unstructured comments (`// ... <EOL>` and `/* ... */`) are discarded.	- Structured comments (`<<** ... **>>`) are translated into Doxygen comments for the corresponding element. - Unstructured comments (`// ... <EOL>` and `/* ... */`) are discarded.	
`package`	`namespace` named after the package - Namespace is nested according to the package hierarchy. - Generated files are stored in a nested directory hierarchy that matches the package hierarchy. - Include guards in generated header files incorporate the package name as a part of the fully qualified interface name (e.g. `ORG_GENIVI_PACKAGE_INTERFACE_H_`).	*Not supported.*	
`import`	`#include` - Included header file(s) contain definition(s) of the referenced symbols. - The path to directory containing the file to be included is derived from the package name.		
`version`	`struct CommonAPI::Version {Major, Minor}` - For Franca `interface`'s, the version is returned by the static method `getInterfaceVersion()` defined in the generated C++ class with the same name as the interface. - For Franca `typeCollection`'s, the version is returned by the static method `getTypeCollectionVersion()` defined in the header file generated for the type collection.	Incorporated into header file names. - Both `interface` and `typeCollection` header files end with the suffix `"*_v<major>_<minor>.h"` - Forwarding header is generated for accessing the most recent known version of client-side bindings (`"if_*_v<major>.h"`).	
`const`	*Not supported.*	*Not supported.*	
Primitive types			
`Int8/16/32/64`	`int8/16/32/64_t`	`int8/16/32/64_t`	`int8/16/32/64_t`
`UInt8/16/32/64`	`uint8/16/32/64_t`	`uint8/16/32/64_t`	`uint8/16/32/64_t`
`Integer`	*Not supported.*	*Not supported.*	
`Boolean`	`bool`	`int`	`int` or `bool` for C99+
`Float`	`float`	`float`	`float`
`Double`	`double`	`double`	`double`
`String`	`std::string` UTF-8 encoded; encoding on the wire can be specified through deployment.	`typedef char ipcq_string_t[]` Size restricted to 256 (`IPCQ_STRING_SIZE`) bytes. since version 0.80: char[], maximum size defined in a .fdepl file	

`ByteBuffer`	`std::vector<uint8_t>`	`typedef char ipcq_buffer_t[]` Size restricted to 4096 (`IPCQ_BYTE_BUFFER_SIZE`) bytes. since version 0.80: unsigned char[], maximum size defined in a .fdepl file	
Derived types			
`array _ of T`	`std::vector<T>`	`typedef T _[]` Size restricted to 256 (`IPCQ_ANONYMOUS_ARRAY_SIZE`) elements. since version 0.80: maximum size defined in a .fdepl file	
`enumeration _`	`enum class _` Based on `int32_t` by default, but can be overridden through deployment.	`typedef enum {} _`	
`struct _`	`struct _: CommonAPI::SerializableStruct` - `polymorphic` qualifier results in `CommonAPI:: PolymorphicSerializableStruct` as the base class.	`typedef struct {} _` - `polymorphic` qualifier is not supported.	
`union`	`CommonAPI::Variant<>`	*Not supported.*	
`map K to V`	`std::unordered_map<K, V>`	*Not supported.*	
`typedef`	`typedef`	`typedef`	
Type collection definition			
`typeCollection`	Header file that defines `typeCollection` elements. - Header file is named either after the `typeCollection` or "`AnonymousTypeCollection.h`". - Header file is stored in the directory structure that matches the containing package hierarchy for `typeCollection`. - The elements are put into the `namespace` that is named after the `typeCollection` and is nested in its package namespace.	Header file that defines `typeCollection` elements. - Header file is named after the `typeCollection`. - *Anonymous* `typeCollection`'s *are not supported.* - The `typeCollection` name is used as a prefix for its element identifiers.	
Interface definition			
`interface X`	`class X` - Provides access to basic interface metadata (interface ID and version) and defines encapsulated data types including method errors. - Types of the client proxy and server stub are defined as `class XProxyBase` and `class XStub` respectively.	Header file that defines `interface` elements - *Bindings independent of middleware are not supported.* All generated code assumes usage of IPCQ as the middleware. - Header file is named after the `interface` ("`t_X_*.h`"). - Additional header files are generated to define bindings for clients ("`if_X_*.h`") and servers ("`srv_if_X_*.h`").	
`attribute T _`	- On the client side, implemented by default by `CommonAPI::*Attribute<T> _` No qualifier maps to `ObservalbleAttribute<>`; `readonly` maps to `ObservableReadonlyAttribute<>`; `noSubscriptions` maps to `Attribute<>`; `readonly noSubscriptions` maps to `ReadonlyAttribute`. - On the server side, implemented by default as `T <_>AttributeValue_` In general, the server must implement the methods for getting, setting, validating the value of each individual attribute.	- On the client side, implemented by accessor functions `ipc_<interface>_get_<attribute>()` retrieves the value; `ipc_<interface>_rcv_<attribute>()` retrieves the values and registers for the next update notification (which must be accessed by regularly re-registering); *attribute setting by clients is not supported.* - On the server side, implemented by locking and notification infrastructure `ipc_<interface>_set_<attribute>()` sets the value	

`method`	`virtual` method of a `class` - On the client side, `in` parameters are passed as `const` reference and `out` parameters are passed by reference. For each method, a synchronous and asynchronous (ending with `Async` suffix) versions are generated. Synchronous version takes all `in` and `out` arguments and additionally parameters `CommonAPI::CallStatus` and – if `error` clause is specified – `<method-name>Error`, both by reference. Asynchronous version takes only `in` arguments and an additional parameter `<method-name>AsyncCallback`. - On the server side, `in` parameters are passed by value and `out` parameters are passed by reference. Stubs do not differentiate between synchronous and asynchronous calls. Each method takes an additional argument of type `std::shared_ptr<CommonAPI::ClientID>` that is implemented in a middleware-specific way and is used to identify the caller. If method has an `error` clause, `<method-name>Error` is passed by reference. - Qualifier `fireAndForget` results in omitted implementation of asynchronous invocation for on the client side. On the server side, there is no impact. - `error` clause on methods is implemented by an `enum class`.	accessor functions - `in` and `out` parameters are passed aggregated into `struct`'s. - On the client side, both synchronous (`ipc_<interface>_call_<method>()`) and asynchronous (`ipc_<interface>_call_<method>_async()`) invocations are supported. - On the server side, the infrastructure to fetch `in` parameters and post the method results is provided. - `error` clause on methods is implemented by `enum ipc_<interface>_<method>_err_t;` *extension of method error types is not supported.*	
`broadcast`	- On the client side, implemented by `CommonAPI::*Event<>` No qualifier maps to `Event<>`; `selective` maps to `SelectiveEvent<>`. - On the server side, implemented by `fire...()` method for sending the signal Qualifier `selective` results into an additional attribute of type `CommonAPI::ClientIdList` for each signal to track its subscribers. It is possible to deny subscriptions.	*Not supported.*	
`extends`	type-dependent - `interface` inheritance is implemented as `class` inheritance and allows overloading methods and broadcasts (by supporting different signatures for the same feature name). - `enum` inheritance is implemented by including references to parent's definition into the child. - `struct` inheritance is implemented as `struct` inheritance. - `union` inheritance is implemented by duplicating the parent's definition in the child.	*Not supported.*	
`manages`	- On the client side, `CommonAPI::ProxyManager` - On the server side, methods to register the stub implementation of the managed interface and to track its instances.	*Not supported.*	